CSE202
1 - Divide and Conquer

TA: Joseph L.

I. General Concepts

- **Divide-and-conquer**: A class of algorithms in which one recursively splits sub-problems into those of smaller input size, and then later recombines the sub-problems' solutions into a solution for the original problem.
  - Splitting should end on a **base case** – a fixed-sized input for which the solution can be easily computed.
  - A complete, high-level description of the algorithm should describe the base case, means of splitting, means of recombination, and time complexity.
- **Master Theorem**: A theorem that quickly derives the complexity for a recurrence of the form $T(n) \le aT\left(\frac{n}{b}\right) + f(n)$, where $f(n) = O(n^c)$ and $T(1) = 1$. This form is common among divide-and-conquer algorithms.

  polynomial

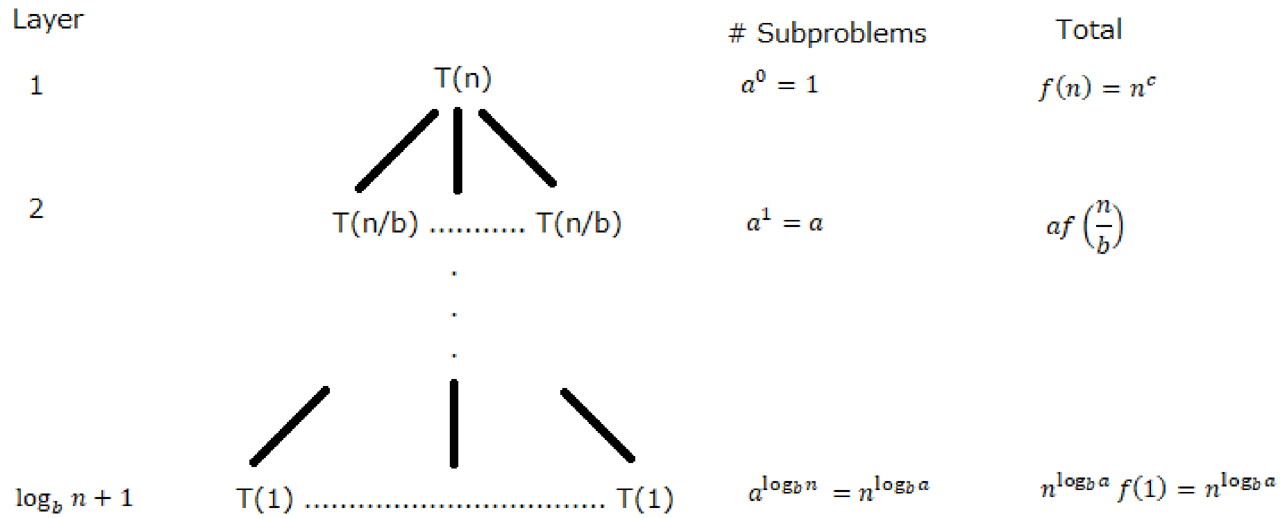

Figure 1. A general recurrence tree.

  - Key log property: $a^{\log_b n} = n^{\log_b a}$
  - The term $f(n)$ can be thought of as the complexity required to *combine subproblems*.
  - This theorem involves **three** cases, depending on the relative values of $\log_b a$ and c.
    - Case 1: $c < \log_b a$
      - The work done is *bottom-heavy*. The lower layers are of higher polynomial degree than the higher ones, and thus dominate the asymptotic runtime.
      - E.g., the last layer has $O\left(n^{\log_b a}\right)$, and thus has a higher degree than the top layer with $O(n^c)$ since $\log_b a > c$.
      - Therefore, $\boldsymbol{T(n) = O\left(n^{\log_b a}\right)}$.
    - Case 2: $c = \log_b a$
      - The work done is distributed evenly across layers. Each layer has complexity $O(n^c)$ since a given layer j has a^j subproblems, each of complexity $O\left(\left(\frac{n}{b^j}\right)^c\right)$. To elaborate:
        - $c = \log_b a$ can be rewritten as $b^c = a$.
        - Substitute: $a^j \left(\frac{n}{b^j}\right)^c = (b^c)^j \left(\frac{1}{(b^c)^j}\right) n^c = n^c$.

        - There are $O(\log_b n)$ such layers.
        - Therefore, $\boldsymbol{T(n) = n^c \log_b n}$.
    - Case 3: $c > \log_b a$
        - The work done is *top*-heavy. The higher layers are of higher polynomial degree than the lower ones, and thus dominate the asymptotic runtime.
        - E.g., the first layer has $O(n^c)$, which has higher polynomial degree than the last layer with $O(\log_b a)$, since $c > \log_b a$.
        - Therefore, $\boldsymbol{T(n) = O(n^c)}$.
  - Takeaways:
    - The Master Theorem is useful for quickly figuring out the complexity for a recurrence of the above form.
    - You may use the Master Theorem, but be sure to understand how each case is derived.
- Proofs of Correctness
  - The piecewise nature of divide-and-conquer lends itself naturally to proof by induction. Recall that a mathematical induction consists of the following parts:
    - **Base Case**: Prove that your algorithm works correctly for some basic case m_0. Inputs falling under the base case should have sizes at or below a *small constant* – e.g., leaf nodes in the above recurrence tree.
    - **Induction Hypothesis**: Assume your algorithm works correctly for *all* inputs up to an arbitrary step $\hat{m}$ – e.g., nodes of the $\hat{m}$th layer of the above recurrence tree, counting from the bottom.
    - **Induction Step**: Prove that, assuming your induction hypothesis holds, your algorithm correctly performs the next step $\hat{m} + 1$ - e.g., the nodes of layer $\hat{m} + 1$ in the above recurrence tree.
    - **Conclusion**: Formally state that because your base case, induction hypothesis, and induction step hold, your algorithm is wholly correct for some defined set of inputs. In our ongoing example, our base case was m_0 and we worked our way up, so our algorithm works for *every* $m \geq m_0$ – i.e., every layer of our recurrence tree.
  - Other methods of proof may be useful for proving portions of your induction (namely, your induction step).

II. Mergesort

- *Description*: We would like to efficiently sort a list of size n.
- *Splitting*: For a given list k, split into two lists i, j of equal size.
- *Base Case*: A list of size $n \leq 2$ can be sorted through simple comparison, in $O(1)$ time.
- *Combining*: Simply merge the two sorted sub-lists, in $O(n)$ time, using the merging algorithm detailed in section 2 of K&T.
- *Complexity*: $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$
  - Master Theorem, Case 2. Complexity is $\boldsymbol{O(nlogn)}$.
- *Correctness, by Induction*
  - *Base Case*: Sub-problems at leaf nodes have lists of size 1 or 2, and are thus trivially, correctly sorted.
  - *Induction Hypothesis*: Suppose that all sublists, up to an arbitrary size m, are correctly sorted. Let i, j be two such sublists.
  - *Induction Step*: We would like to prove that, if the hypothesis holds, the combined list k (of size approximately $2m$) is correctly sorted for i, j.
    - The merging algorithm works correctly for sorted lists, as proven in the book.
    - By the induction hypothesis, i, j are indeed sorted, thus satisfying the above precondition. Thus, k is sorted.

- *Conclusion*: Our base case, hypothesis, and step hold. Therefore, by induction, our algorithm works for lists of all sizes $n \geq 0$.

- Notes
  - $O(nlogn)$ is the best possible runtime for comparison-based sorting. This is important to remember for this class, as many problems will involve sorting as a subroutine or precomputation.
  - In the induction step, we cited without proof an algorithm that was proven in the book. In this class, you are allowed to re-use a previously discussed algorithm without proof *if* your solution uses it *directly*. If you instead use a variant of a discussed algorithm, then you must prove it from scratch.
- Related Problems: Counting inversions

III. Finding the closest pair of points on a 2D plane

- *Description*: Given n points on a plane, we would like to find the pair with the smallest distance between the two points.
- Assume, without loss of generality, that no two points share the same x or y coordinate. Whenever this is not satisfied, the coordinate plane can be rotated until it is satisfied.
- *Notation*:
  - Set of points $P = \{p_1, \dots, p_n\}$
  - $d(p_i, p_j)$ is the Euclidean distance between $p_i, p_j \in P$.
- *Preprocessing*:
  - Let P_x be P, sorted by x-coordinates.
  - Let P_y be P, sorted by y-coordinates.
  - This preprocessing can be done in $O(nlogn)$ using mergesort.
- *Splitting*: For a given set of points P, split into two sets l, r of equal size using a vertical line.

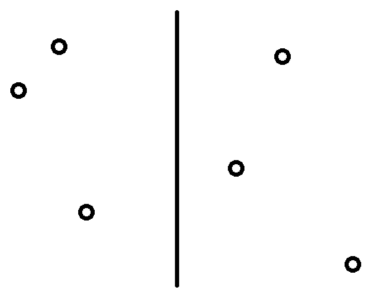

Figure 2. Points are split by a vertical line.

- *Base case*: For a set of ≤ 3 points, the closest pair can be found in constant time through pairwise comparison.
- *Combining*: For a set $S = l \cup r$, the closest pair is one of the following:
  - The closest pair in l. Let us refer to this as (q_0^*, q_1^*). It goes without saying that since S contains all points from l, this pair is a valid candidate for the closest pair in S.
  - Analogously, the closest pair in r. Let us refer to this as (r_0^*, r_1^*).
  - A pair containing one point from each set. We would like to determine this efficiently.
    - Let $\delta = \min\left(d(q_0^*, q_1^*), d(r_0^*, r_1^*)\right)$. We are interested in whether there exist pairs closer than δ, with one point from each set.
    - Let us consider a 2δ-by-2δ region containing at least one set of candidate points, as in Figure 4.

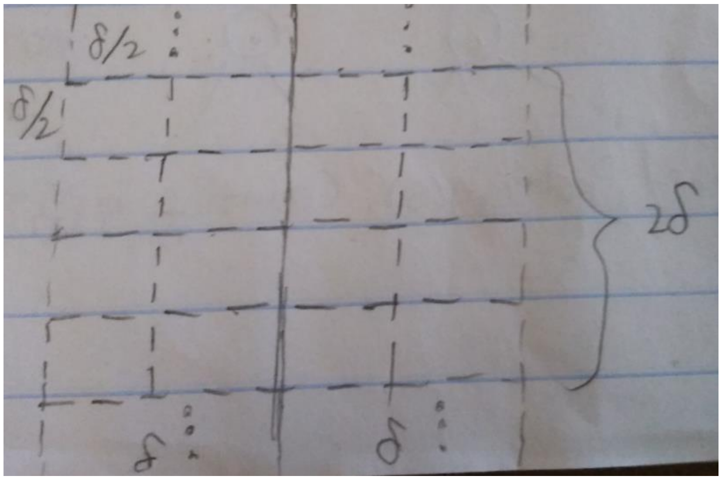

Figure 3. A 2δ-by-2δ region, split into boxes of dimensions $\frac{\delta}{2}$-by-$\frac{\delta}{2}$. The left half contains points in l, and the right half contains points in r.

- We first argue that each of the above boxes can contain *at most one* point.
  - Suppose there exists a box containing two points. Then, the distance between this pair of points is at most $\frac{\delta}{\sqrt{2}}$ (by trigonometry, if the points are on opposite corners of the box), which is smaller than δ. This contradicts our definition of δ as the *smallest distance between points on the same side*. Thus, by proof by contradiction, there cannot exist a box with more than one point.
- Let S_y be the points in S, sorted by y-coordinates. This can be derived from the pre-computed P_y in $O(n)$ time. We argue that any pair of points (p_l, p_r) s.t. $p_l \in l$ and $p_r \in r$ and $d(p_l, p_r) < \delta$, is *at most 15 positions apart* in S_y.
  - Suppose there exists a pair that is more than 15 positions apart, with $d(p_l, p_r) < \delta$. Because we established that each box has at most one point, this requires p_l, p_r to be more than 3 rows of boxes apart. This in turn means $d(p_l, p_r) > 3\left(\frac{\delta}{2}\right)$, which is larger than δ, thus violating our condition that $d(p_l, p_r) < \delta$. Thus, by proof by contradiction, p_l, p_r are at most 15 positions apart in S_y.
- Thus, each point in S has ≤ 15 candidates with which it may form the closest pair – the number is bounded by a *constant*, allowing us to check all candidates in $O(1)$ time. Because our algorithm is exhaustive, it is guaranteed to find the closest pair, if it exists.
- We check the ≤ 15 pair candidates of up to $O(n)$ points. Each set of checks is $O(1)$. Thus, our complexity for combining is $O(n)$.

- *Complexity*: $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$
  - Master Theorem, Case 2. The complexity is $\boldsymbol{O(nlogn)}$.
  - The preprocessing is also $O(nlogn)$, so our total complexity is the sum: $\boldsymbol{O(nlogn)}$.
- *Correctness by induction*
  - As is often the case with divide-and-conquer, we've already proven, in our description of the algorithm, most pieces of our proof by induction. Our proof will thus largely restate this prior work.
  - *Base Case*: For a set of ≤ 3 points, we use pairwise comparison. As this exhaustively tries all possible pairs, it is clearly correct.
  - *Induction Hypothesis*: Suppose that for all partitions smaller than or as large as some partition l, r, we have successfully computed the closest pairs.

- *Induction Step*: We would like to prove that, if our hypothesis holds, our algorithm correctly computes the closest pair for $l \cup r$. We have already proven the correctness of our "combining" step above, so we simply restate that fact.
- *Conclusion*: Our base case, hypothesis, and step hold. Therefore, our algorithm is correct for all sets of points of size $n \geq 2$ (the problem is about finding a pair, and is thus ill-posed for $n < 2$).

- Notes
  - When finding the closest pair spanning l, r, we can save some effort by eliminating all points that fall a distance $\geq \delta$ from the splitting line. This does not change the asymptotic runtime, but does eliminate unneeded effort.
  - This problem generalizes naturally to higher dimensions. In general, higher-dimensional variants are not efficiently solvable with divide-and-conquer.

IV. Integer Multiplication

- *Description*: We would like to multiply two numbers, each represented with up to n bits, efficiently.
- Introduction
  - Let x, y be two n-bit numbers – e.g., $n \geq \log_2 x$ and $n \geq \log_2 y$.
  - The elementary school algorithm for multiplying x, y is quadratic w.r.t. the number of digits.
    - Recall that you do one single-digit multiplication per pair of digits.
  - The same algorithm can be applied in binary, where it is quadratic with respect to the number of bits – i.e., it is $O(n^2)$.
  - As it turns out, there exists a divide-and-conquer algorithm of lower polynomial runtime.
- *Splitting*: We can rewrite x and y in terms of their higher and lower bits.
  - Let x_1 be the $\frac{n}{2}$ higher bits and x_0 be the $\frac{n}{2}$ lower bits. We can now rewrite x as follows:
$$x = x_1 * 2^{\frac{n}{2}} + x_0$$
  - Rewrite y analogously.
  - Now, we can rewrite the product xy as follows:
$$xy = \left(x_1 * 2^{\frac{n}{2}} + x_0\right)\left(y_1 * 2^{\frac{n}{2}} + y_0\right) = x_1y_1 * 2^n + (x_1y_0 + x_0y_1) * 2^{\frac{n}{2}} + x_0y_0$$
  - The rightmost expression seemingly has *four* products, of $\frac{n}{2}$ bits each, that must be computed:
    - x_1y_1
    - x_1y_0 $T(n) = 4T(\frac{n}{2}) + O(n)$
    - x_0y_1
    - x_0y_0
  - Note, however, that we do not directly need the values of x_1y_0 and x_0y_1 - we only need $x_1y_0 + x_0y_1$. We will derive this value based on x_1y_1 and x_0y_0 (which we do need), as well as *one* other multiplication, for a total of *three*.
    - Write: $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$.
    - Rewrite $x_1y_0 + x_0y_1$ as follows:
$$x_1y_0 + x_0y_1 = (x_1 + x_0)(y_1 + y_0) - (x_1y_1 + x_0y_0)$$
    - Now, we need only 3 products and thus 3 subproblems:
      - $(x_1 + x_0)(y_1 + y_0)$ addition is cheaper than multiplication
      - x_1y_1
      - x_0y_0
- *Base Case*: For $n = 1$, the numbers can simply be multiplied together.
- *Combining*: Given the products computed in our subproblems, we simply need to apply 2 bit-shifts in $O(n)$ time and add 3 numbers together, also in $O(n)$ time.

- *Complexity*: $T(n) \le 3T\left(\frac{n}{2}\right) + O(n)$ $\log_b a = \log_2 3 > 1$
  - Master Theorem, case 1. The complexity is $\boldsymbol{T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})}$.
- *Correctness by induction*
  - Again, we've already proven most of what's needed here.
  - *Base Case*: Multiplication of numbers with a bounded number of bits (e.g., 1) can be done correctly and in constant time.
  - *Induction Hypothesis*: Suppose all products are correctly computed up to an arbitrary bit size m (where m is a power of 2, due to how our splitting/recombining works).
  - *Induction Step*: We would like to prove that, if our hypothesis holds, then our algorithm correctly combines the solutions of size m bits into solutions of size $2m$ bits. As the mathematics in our "splitting" and "combining" sections are algebraically sound, we simply restate that fact.
  - *Conclusion*: Our base case, hypothesis, and step hold. Therefore, our algorithm works correctly for all numbers sized $n \ge 1$ bits.

V. Convolution with Fast Fourier Transforms

- *Description*: We would like to efficiently convolve two vectors of length n.
- Background
  - **Convolution** of vectors a, b
    - Inputs: Vectors a of length m, b of length n
    - Output: Vector c of length $m + n - 1$.
      - $c = a * b$
      - $c_k = \sum_{i,j:i+j=k} a_i b_j$, where $0 \le i \le m$ and $0 \le j \le n$.
    - An application: multiplying polynomials $A(x)$ and $B(x)$.
      - Write $A(x)$ and $B(x)$ as vectors of coefficients:
        - $A(x) = a_0 + a_1 x + \cdots + a_{m-1}x^{m-1}$ as $a = (a_0, a_1, \dots, a_{m-1})$
        - $B(x) = b_0 + b_1 x + \cdots + b_{m-1}x^{m-1}$ as $b = (b_0, b_1, \dots, b_{m-1})$
      - If $C(x) = A(x)B(x)$, then $c = a * b$.
    - For simplicity in this section, assume $m = n$. This is without loss of generality since one can always pad the smaller vector with 0's.
    - A naïve implementation of convolution achieves complexity $O(n^2)$ by multiplying all a_i, b_j pair-wise. We will discuss an $O(nlogn)$ algorithm.
  - **Complex $\boldsymbol{k}$th roots of unity**
    - Recall the unit circle on the complex plane:

$$C = A(x)B(x)$$
$$= (a_0 + a_1 x + \cdots + a_{m-1} x^{m-1})(b_0 + b_1 x + \cdots + b_{m-1} x^{m-1})$$
$$= a_0 b_0 + a_1 b_0 x + \cdots + a_{m-1} b_0 x^{m-1} + a_0 b_1 x + a_1 b_0 x^2 + \cdots + a_{m-1} b_1 x^m$$
$$+ a_0 b_{m-1} x^{m-1} + a_1 b_{m-1} x^m + \cdots + a_{m-1} b_{m-1} x^{2m-2}$$
$$= a_0 b_0 + (a_1 b_0 + a_0 b_1) x + \cdots$$
$$c_i = \sum_{0 \le j \le i} a_j b_{i-j}$$

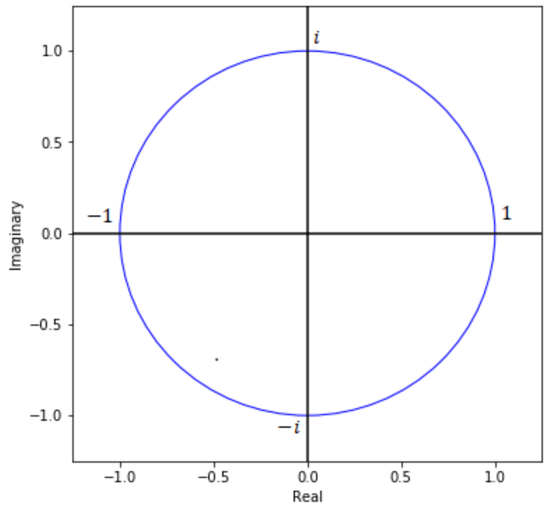

Figure 4. The unit circle on the complex plane.

- A point on this unit circle can be written in polar coordinates as $e^{i\theta}$, where $\theta \in [0,2\pi)$.
- The complex kth roots of unity are the k equally-spaced (in terms of angle) points $e^{2\pi ji/k}$, for $j \in \{0,1,\dots,k-1\}$, that satisfy $\left(e^{i\theta}\right)^k = 1$.

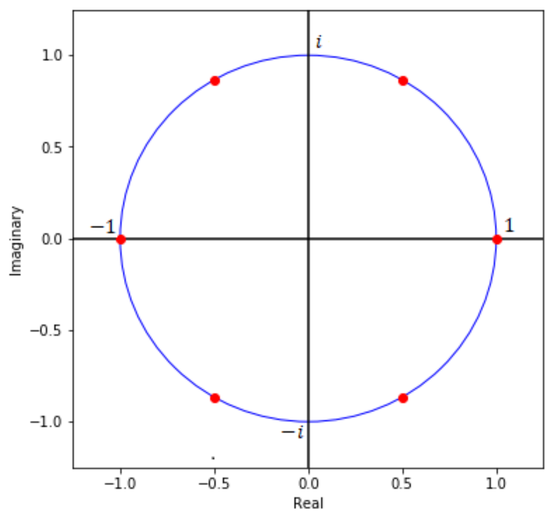

Figure 5. The complex 6th roots of unity are shown in red.

- **Polynomial interpolation**
  - Recall that a degree-d polynomial can be recovered from interpolating $d+1$ points that lie on it.
  - This procedure takes $O(d)$ time.
- Finally, onto the algorithm…

- Key ideas
  - $C(x) = A(x)B(x)$ can be recovered from interpolating $\sim 2n$ points that lie on $A(x)B(x)$.
  - $C(x_j) = A(x_j)B(x_j)$ for any x_j. Thus, we would like to evaluate both A and B on $\sim 2n$ points, and then multiply their values together.
  - Evaluating $A(x)$ on one point takes $O(n)$ time (it's essentially a vector dot product). Naively evaluating on $2n$ points would therefore be $O(n^2)$, and thus yield no improvement.
  - **Key point**: A $\frac{k}{2}$th root of unity is also a kth root of unity. Thus, if we evaluate our polynomials on the kth roots of unity, we can reuse our work for the $2k$th roots of unity. We now see the potential for a divide-and-conquer strategy.

- *Splitting*
  - The remaining sections will cover how to evaluate the polynomial A on the points; an identical procedure is also applied to B.
  - Split $A(x)$ into its odd and even coefficients:
    - $A_{even}(\hat{x}) = a_0 + a_2\hat{x} + a_4\hat{x}^2 + \cdots + a_{n-2}\hat{x}^{(n-2)/2}$
    - $A_{odd}(\hat{x}) = a_1 + a_3\hat{x} + a_5\hat{x}^2 + \cdots + a_{n-1}\hat{x}^{(n-2)/2}$
    - $A(x) = A_{even}(x^2) + xA_{odd}(x^2)$
  - We now have $A_{even}(\hat{x})$ and $A_{odd}(\hat{x})$ - two polynomials of halved degree, and thus two subproblems.
  - For a given subproblem, we will evaluate A_{even} and A_{odd} on the nth roots of unity.
- *Base Case*: We can compute $A(x)$ directly on the 2nd roots of unity.
- *Combining*: Given the nth roots of unity, we need to evaluate $A(x)$ on the $(2n)$th roots of unity. Let $\omega_{j,2n} = e^{2\pi ji/2n}$ be one such root.
  - $\left(e^{\frac{2\pi ji}{2n}}\right)^2 = e^{2\pi ji/n}$, with is an nth root of unity.
  - $A(\omega_{j,2n}) = A_{even}(\omega_{j,2n}^2) + \omega_{j,2n}A_{odd}(\omega_{j,2n}^2) = A_{even}(\omega_{j,n}) + \omega_{j,n}A_{odd}(\omega_{j,n})$
  - Both terms in the rightmost expression were computed in the subproblems. Therefore, it is $O(1)$ to evaluate $A(\omega_{j,2n})$ for one value of j, and $O(n)$ for all. Combining is $O(n)$.
- *Complexity*: $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$ $a=2, b=2, c=1$
  - Master Theorem, Case 2: complexity is $O(nlogn)$.
  - We perform the exact same steps for B: we spend another additive $O(nlogn)$.
  - Computing C at the end is an additive $O(n)$.
  - Total: $O(nlogn + nlogn + n) = \boldsymbol{O(nlogn)}$
- *Correctness (of the evaluations on A, B) by induction*
  - *Base Case*: Our base case is a direct polynomial evaluation, and is thus correct.
  - *Induction Hypothesis*: Suppose A_{even}, A_{odd} are correctly computed up to some degree n.
  - *Induction Step*: Prove that $A(x)$ is correctly computed, given the hypothesis, for the $(2n)$th roots of unity.
    - $A(x) = A_{even}(x^2) + xA_{odd}(x^2)$ is algebraically sound.
    - $(2n)$th roots of unity, when squared, produce nth roots of unity. Therefore, our logic for reusing past computations is algebraically sound.
  - *Conclusion*: Our hypothesis and step are correct, so by induction, our algorithm is correct for all possible values of n.
- Notes
  - One useful skill is the ability to recognize problems that can be modeled as convolutions.

VI. Sample Problem

We will do the following problem from Chapter 5 of K&T [1], condensed and paraphrased below.
Description: There are n bank cards and, while we can't read their numbers directly, we can compare two cards in $O(1)$ time. Among these n cards, is there a set of *more than* $\frac{n}{2}$ of them that are all equivalent to another? Solve this problem in $O(nlogn)$ time.

Input: n bank cards
Output: A set of $> \frac{n}{2}$ equivalent bank cards (if any)

Observations: We can compare cards pair-wise, which suggests that our method of splitting should perhaps be by halves. If we can get a recurrence tree with $O(logn)$ layers, with each one doing a total of $O(n)$ work, then we can achieve our desired runtime of $O(nlogn)$.

Splitting: Recursively split our list of cards into halves. Each subproblem will consist of the following information:

- Which card, if any, makes up more than half of its list? This is the "solution" of our subproblem.
- Exactly how many copies of that card exist in the list?

Base Case: For $n \leq 3$ cards, we can do a fixed number of comparisons to answer both of the above questions.

Combining: Given two halves l and r and their respective solutions s_l and s_r (which can both be null), what cards are candidates for the solution s_p of the combined list p? We argue that the only options are s_l and s_r.

- *Claim*: The solution of a combined problem p (of size n), if it exists, comprises more than half of at least one of l or r - that is, there are *more than* $\frac{n}{4}$ instances of it in either list.
- *Proof*: Suppose neither list has more than $\frac{n}{4}$ instances of s_p, the solution of p. Then, combined, p can have at most $\frac{n}{2}$ instances of s_p. However, we only consider a card to be a solution if it comprises *more* than half of the list; thus, s_p wouldn't be a solution in this case. This contradicts our knowledge that s_p is a solution and thus, by proof by contradiction, we now know that at least one of l, r must contain *more than* $\frac{n}{4}$ instances of s_p.

Thus, because s_l and s_r are the only cards that comprise more than half of their respective lists, they are indeed the only candidates for s_p. With this knowledge, we simply figure out, in $O(n)$ time, which (if either) of them comprises more than half of p. Note that if $s_l = s_r$, then $s_l = s_r = s_p$ without any further inspection.

Complexity: $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$.

- *Master Theorem*, Case 2. The complexity is the desired $O(nlogn)$.

Correctness: Once again, with induction
Base Case: For $n \leq 3$ cards, we compare pair-wise, so that is trivially correct.

Induction Hypothesis: Suppose we have correctly computed solutions for subproblems of size up to n.

Induction Step: We would like to prove that if the hypothesis is true, we can correctly solve subproblems of size up to $2n$. We did this in the "Combining" section, so we refer to that.

Conclusion: Our hypothesis and step are correct so, by induction, our algorithm is correct for all possible values of n.

References

1. Kleinberg, Jon, and Éva Tardos. *Algorithm Design*. Boston: Pearson/Addison-Wesley, 2006. Print.