Divide and Conquer

CMSC 27230: Honors Theory of Algorithms

January 20 - January 27, 2023

Corresponding section(s) of Kleinberg-Tardos: 5.1,5.2,5.3,5.4,5.6

1 Overview

Divide and conquer algorithms solve problems by splitting them into smaller subproblems and then cleverly combining the solutions for the subproblems into a solution for the entire problem.

2 Recursive Algorithms and Recurrence Relations

By their nature, divide and conquer algorithms are usually recursive algorithms which call themselves. To analyze the performance of recursive algorithms, we let $T(n)$ be the time the algorithm takes when the input has size n and write down a recurrence relation for $T(n)$. We will then try and solve this recurrence relation.

Example 2.1 (Mergesort)**.** *The mergesort algorithm sorts an array of n elements as follows: If $n = 1$, the array is already sorted. Otherwise, we take the following steps:*

1. *Mergesort elements 1 through $\lfloor \frac{n}{2} \rfloor$ of the array.*
2. *Mergesort elements $\lfloor \frac{n}{2} \rfloor + 1$ through n of the array.*
3. *If $A = (a_1, \ldots, a_{n_1})$ and $B = (b_1, \ldots, b_{n_2})$ are two sorted sequences of elements then we can merge these sequences into a sorted sequence $C = (c_1, \ldots, c_{n_1+n_2})$ as follows:*

   Initialization: Start with $i = 1$ and $j = 1$.

   Iterative step: If $j > n_2$ or $i \leq n_1$ and $a_i < b_j$ then set $c_{i+j-1} = a_i$ and increase i by 1. If $i > n_1$ or $j \leq n_2$ and $b_j \leq a_i$ then set $c_{i+j-1} = b_j$ and increase j by one.

   Using this algorithm, we merge the two sorted halves of the array into the full sorted array.

Up to a constant factor, the recurrence relation for mergesort is $T(n) = 2T(\frac{n}{2}) + n$

Example 2.2 (Binary search)**.** *The binary search algorithm checks if an element x is in a sorted array $A = (a_1, \ldots, a_n)$ as follows:*

Input to the recursive algorithm: The recursive algorithm requires indices i and j such that if x is in A, it must be after the ith element of A and before the jth element of A.

Recursive algorithm: If $j - i \leq 1$ then x cannot be in A and we reject. Otherwise, take $k = \lfloor \frac{i+j}{2} \rfloor$. We now have the following cases:

(a) *If $a_k = x$ then we have found x.*

(b) *If $a_k < x$ then we use binary search with $i' = k$ and $j' = j$.*

(c) *If $a_k > x$ then we use binary search with $i' = i$ and $j' = k$.*

To check if an element x is in A, we initially call the recursive algorithm on A with $i = 0$ and $j = n + 1$.

Up to a constant factor, the recurrence relation for binary search is $T(n) = T(\frac{n}{2}) + 1$

3 Solving recurrence relations

We now describe two ways of solving recurrence relations. The first way is to expand out the recurrence relation. The second way is to take an educated guess, check it, and adjust accordingly.

3.1 Expanding out recurrence relations

One way to solve recurrence relations is to iteratively substitute the recurrence relation into itself. This method is direct and intuitive, but it may not work as well for more complicated recurrence relations.

Example 3.1. *Consider the recurrence relation $T(n) = 2T(\frac{n}{2}) + n$ for mergesort. Repeatedly substituting this recurrence relation into itself, we have that*

1. $T(n) = 2T(\frac{n}{2}) + n$
2. $T(n) = 4T(\frac{n}{4}) + 2\frac{n}{2} + n$
3. $T(n) = 8T(\frac{n}{8}) + 4\frac{n}{4} + 2\frac{n}{2} + n$
4. $T(n) = 2^k T(\frac{n}{2^k}) + kn$

Now we take $k = log_2(n)$ so that $\frac{n}{2^k} = 1$. This gives us $T(n) = nT(1) + nlog_2(n)$. If $T(1) = 1$ then $T(n) = n + nlog_2(n)$. Thus, mergesort takes $O(nlogn)$ time.

Example 3.2. *Consider the recurrence relation $T(n) = T(\frac{n}{2}) + 1$ for binary search. Repeatedly substituting this recurrence relation into itself, we have that*

1. $T(n) = T(\frac{n}{2}) + 1$
2. $T(n) = T(\frac{n}{4}) + 1 + 1$
3. $T(n) = T(\frac{n}{8}) + 1 + 1 + 1$

4. $T(n) = T(\frac{n}{2^k}) + k$

Now we take $k = log_2(n)$ so that $\frac{n}{2^k} = 1$. This gives us $T(n) = T(1) + log_2(n)$. If $T(1) = 1$ then $T(n) = 1 + log_2(n)$. Thus, binary search takes $O(logn)$ time.

Example 3.3. *Consider the recurrence relation $T(n) = 2T(\frac{n}{4}) + 10$. Repeatedly substituting this recurrence relation into itself, we have that*

1. $T(n) = 2T(\frac{n}{4}) + 10$

2. $T(n) = 4T(\frac{n}{16}) + 2 * 10 + 10$

3. $T(n) = 8T(\frac{n}{64}) + 4 * 10 + 2 * 10 + 10$

4. $T(n) = 2^k T(\frac{n}{4^k}) + 10\left(\sum_{j=0}^{k-1} 2^j\right)$

Now we take $k = log_4(n)$ so that $\frac{n}{2^k} = 1$. Since $2^{log_4(n)} = \sqrt{n}$ and $\sum_{j=0}^{k-1} 2^j = 10(2^k - 1) = 10(\sqrt{n} - 1)$, this gives us that

$$T(n) = \sqrt{n}T(1) + 10(\sqrt{n} - 1) = (T(1) + 10)\sqrt{n} - 10$$

If $T(1) = 1$ then $T(n) = 11\sqrt{n} - 10$

3.2 The homogeneous and inhomogeneous parts of a recurrence relation

In order to describe how to take educated guesses for the solution to recurrence relations, we must first analyze the structre of the solutions we are looking for. Here we consider recurrence relations which are of the form $L(T, n) = g(n)$ where $L(T, n)$ is linear in T.

Definition 3.4. *We say that $L(T, n)$ is linear in T if for all $T_1, T_2 : \mathbb{N} \to \mathbb{R}$ and all $a, b \in \mathbb{R}$*

$$L(aT_1 + bT_2, n) = aL(T_1, n) + bL(T_2, n)$$

Example 3.5. *$L(T, n) = T(n) - 2T(\frac{n}{2})$ is linear in T as for all $T_1, T_2, a, b,$*

$$\begin{aligned} L(aT_1 + bT_2, n) &= aT_1(n) + bT_2(n) - 2aT_1(\frac{n}{2}) - 2bT_2(\frac{n}{2}) \\ &= aT_1(n) - 2aT_1(\frac{n}{2}) + bT_2(n) - 2bT_2(\frac{n}{2}) \\ &= aL(T_1, n) + bL(T_2, n) \end{aligned}$$

Definition 3.6. *Given a recurrence relation of the form $L(T, n) = g(n)$ where $L(T, n)$ is linear in T, we say that $L(T, n)$ is the homogeneous part of the recurrence relation and $g(n)$ is the inhomogeneous part of the recurrence relation.*

To find the general solution to a recurrence relation of the form $L(T, n) = g(n)$ where $L(T, n)$ is linear in T, we must do the following:

1. Find the vector space $\{T_0 : L(T_0, n) = 0\}$ of solutions to the homogeneous part of the recurrence relation.

2. Find a single solution T_1 to the entire recurrence relation $L(T_1, n) = g(n)$.

Proposition 3.7. *$T(n)$ is a solution to the recurrence $L(T, n) = g(n)$ if and only if we can write $T = T_1 + T_0$ where $L(T_0, n) = 0$*

Proof. If $T = T_1 + T_0$ then $L(T, n) = L(T_1, n) + L(T_0, n) = g(n) + 0 = g(n)$. Conversely, if $L(T, n) = g(n)$ then take $T_0 = T - T_1$. Now $T = T_1 + T_0$ and $L(T_0, n) = L(T, n) - L(T_1, n) = g(n) - g(n) = 0$ □

3.3 Educated guessing and checking

We now show how recurrence relations can be solved by taking educated guesses, checking them, and adusting accordingly.

For recurrence relations whose homogeneous part consists of terms of the form $T(cn)$ for some constant c, a good guess is $T_0(n) = n^p$ for some power p. The reason this is a good guess is because $T_0(cn)^p = c^p n^p$. The factors of n^p will cancel and then we just have to chose p so that the constants give us an equality.

To find a single solution to the entire recurrence relation, if $g(n)$ is a multiple of $n^{p'}$ then $T_1(n) = c'n^{p'}$ is often a good guess. The reason for this is that $T_1(cn) = c'c^{p'}n^{p'}$ so the $n^{p'}$ factors cancel and we can try and choose c' to obtain an equality. The main case when this fails is when $n^{p'}$ is a solution to the homogeneous part of the recurrence relation. When this is the case, it turns out that multiplying $g(n)$ by $c'log(n)$ works well.

Example 3.8. *For the recurrence relation $T(n) = 2T(\frac{n}{2}) + n$ for mergesort, we can take $L(T, n) = T(n) - 2T(\frac{n}{2})$ and $g(n) = n$.*

The homogeneous part of this recurrence relation is $L(T_0, n) = 0$ or equivalently $T_0(n) = 2T_0(\frac{n}{2})$. Plugging in $T_0(n) = n^p$ we obtain that $n^p = 2(\frac{n}{2})^p$ which is satisfied when $p = 1$. Thus, the vector space of solutions to the homogenous part is $\{T_0(n) = an : a \in \mathbb{R}\}$.

Here $g(n) = n$ is part of the solution to the homogeneous part so we instead try $T_1(n) = c'nlog_2(n)$. Plugging this in we obtain that $c'nlog_2(n) = 2c'\frac{n}{2}(log_2(n) - 1) + n$ which is true when $c' = 1$. Thus, the general solution to this recurrence relation is

$$T(n) = an + nlog_2(n)$$

If $T(1) = 1$ then solving for a we obtain that $a = 1$ and $T(n) = nlog_2(n) + n$

Example 3.9. *For the recurrence relation $T(n) = T(\frac{n}{2}) + 1$ for binary search, we can take $L(T, n) = T(n) - T(\frac{n}{2})$ and $g(n) = 1$.*

The homogeneous part of this recurrence relation is $L(T_0, n) = 0$ or equivalently $T_0(n) = T_0(\frac{n}{2})$. Plugging in $T_0(n) = n^p$ we obtain that $n^p = (\frac{n}{2})^p$ which is satisfied when $p = 0$. Thus, the vector space of solutions to the homogenous part is $\{T_0(n) = a : a \in \mathbb{R}\}$.

Here $g(n) = 1$ is part of the solution to the homogeneous part so we instead try $T_1(n) = c'log_2(n)$. Plugging this in we obtain that $c'log_2(n) = c'(log_2(n) - 1) + 1$ which is true when $c' = 1$. Thus, the general solution to this recurrence relation is

$$T(n) = a + log_2(n)$$

If $T(1) = 1$ then solving for a we obtain that $a = 1$ and $T(n) = log_2(n) + 1$

Example 3.10. *For the recurrence relation $T(n) = 2T(\frac{n}{4}) + 10$, we can take $L(T, n) = T(n) - 2T(\frac{n}{4})$ and $g(n) = 10$.*

The homogeneous part of this recurrence relation is $L(T_0, n) = 0$ or equivalently $T_0(n) = 2T_0(\frac{n}{4})$. Plugging in $T_0(n) = n^p$ we obtain that $n^p = 2(\frac{n}{4})^p$ which is satisfied when $p = \frac{1}{2}$. Thus, the vector space of solutions to the homogenous part is $\{T_0(n) = a\sqrt{n} : a \in \mathbb{R}\}$.

We now try $T_1(n) = c'$. Plugging this in we obtain that $c' = 2c' + 10$ which is true when $c' = -10$. Thus, the general solution to this recurrence relation is

$$T(n) = a\sqrt{n} - 10$$

If $T(1) = 1$ then the solution is $T(n) = 11\sqrt{n} - 10$

3.4 Advanced remarks

There are a few more things to note.

First, we previously said that we gave a general solution. Actually, this is not quite accurate. The reason is that our initial data only determines $T(n)$ for certain values of n. For example, if $T(n) = 2T(\frac{n}{2}) + n$ then $T(1)$ determines $T(2), T(4), T(8), etc.$ but does not determine $T(n')$ for other values of n'. It is quite possible that T behaves differently on $1, 2, 4, 8, \ldots$ than it does on $3, 6, 12, 24, \ldots$. Thus, strictly speaking we are only finding the general solution for T on a subset of the inputs n. For example, we found the general solution of $T(n) = 2T(\frac{n}{2}) + n$ on the inputs $n = 1, 2, 4, 8, \ldots$ and we found the general solution of $T(n) = 2T(\frac{n}{4}) + 10$ on the inputs $n = 1, 4, 16, 64, \ldots$. We sweep this detail under the rug for this course.

Second, we note that taking $T_0(n) = n^p$ is only a good guess when all of our linear terms have the form $T(cn)$ where c is a constant. If we instead only had terms of the form $T(n + c)$ where c is a constant then taking $T_0(n) = x^n$ is a good guess.

Example 3.11. *The Fibbonacci numbers obey the recurrence relation $F(n) = F(n-1) + F(n-2)$. Plugging in $F(n) = x^n$ we obtain that $x^n = x^{n-1} + x^{n-2}$ which is true if and only if $x^2 - x - 1 = 0$. The two solutions to this equation are $x = \frac{1\pm\sqrt{5}}{2}$. Thus, the general solution to this recurrence relation is $F(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n$.*

To find an explicit expression for the Fibonacci numbers, we can solve for c_1 and c_2 when $F(0) = 0$ and $F(1) = 1$. This gives us $c_1 = \frac{1}{\sqrt{5}}$ and $c_2 = -\frac{1}{\sqrt{5}}$ so the Fibonacci numbers are $F(n) = \frac{1}{\sqrt{5}}\left((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n\right)$

If we have different kinds of terms or a mixture of these kinds of terms, then a different guess may be needed or it may be impossible to give an explicit solution for the recurrence relation.

Third, if the recurrence relation has terms which depend in a non-linear way on T then the general solution no longer has the form $T = T_1 + T_0$ and it is generally difficult or impossible to find the general solution to such recurrence relations.

Example 3.12. *The recurrence relation $T(n) = T(n-1)^2$ is not linear in n because $T(n-1)$ is squared. One solution to this recurrence relation is $T(n) = 2^{2^n}$ as $(2^{2^{n-1}})^2 = 2^{2 \cdot 2^{n-1}} = 2^{2^n}$. However, the general solution to this recurrence relation is not $c2^{2^n}$. Instead, the general solution to this recurrence relation is $T(n) = c^{2^n}$ as $(c^{2^{n-1}})^2 = c^{2 \cdot 2^{n-1}} = c^{2^n}$*

4 Counting Inversions

Problem 4.1. *In the counting inversions problem, we are given a sequence $a_1, \dots, a_n$ of distinct numbers. We are then asked how many pairs $i < j \in [n]$ there are such that $i < j$ but $a_i > a_j$.*

Example 4.2. *For the sequence $2, 5, 3, 1, 4$ there are 5 inversions. $a_1 = 2 > a_4 = 1$, $a_2 = 5 > a_3 = 3$, $a_2 = 5 > a_4 = 1$, $a_2 = 5 > a_5 = 4$, and $a_3 = 3 > a_4 = 1$.*

Naïvely, solvig this problem would take $O(n^2)$ time as we may have to consider every pair i, j.

Algorithm 4.3. *Given an array A of n distinct elements, we can simultaneously count the number of inversions in A and sort A as follows:*
Base case: If $n = 1$, the array is already sorted and there are no inversions.
Otherwise, we take the following steps:

1. *Recursively apply this algorithm to elements 1 through $\lfloor \frac{n}{2} \rfloor$ of A and let $count_L$ be the number of inversions that were in these elements of A.*

2. *Recursively apply this algorithm to elements $\lfloor \frac{n}{2} \rfloor + 1$ through n of A and let $count_R$ be the number of inversions that were in these elements of A.*

3. *If $A = (a_1, \dots, a_{n_1})$ and $B = (b_1, \dots, b_{n_2})$ are two sorted sequences of elements then we can count the number of pairs i, j such that $a_i > b_j$ and merge these sequences into a sorted sequence $C = (c_1, \dots, c_{n_1+n_2})$ as follows:*

   Initialization: Start with $i = 1$, $j = 1$, and $count_{between} = 0$.

   Iterative step: If $j > n_2$ or $i \leq n_1$ and $a_i < b_j$ then set $c_{i+j-1} = a_i$ and increase i by 1. If $i > n_1$ or $j \leq n_2$ and $b_j \leq a_i$ then set $c_{i+j-1} = b_j$, increase $count_{between}$ by $n_1 - i + 1$, and increase j by one.

   After applying this algorithm to merge the two sorted havles of A, we have sorted A and the total number of inversions in A was $count_L + count_R + count_{between}$

To see why this merging algorithm successfully counts the number of inversions, we can think of it as follows. At each step, we have the elements in C followed by the remaining elements of A followed by the remaining elements of B. When an element of A is added to C, nothing changes. When an element b_j of B is added to C, $b_j < a_{i'}$ for all $a_{i'}$ remaining in A, so each of these remaining $a_{i'}$ was in an inversion with b_j. Shifting b_j before all of these $a_{i'}$ removes all of these inversions, reducing the number of inversions by $n_1 - i + 1$, the number of elements remaining in A. Thus, $count_{between}$ keeps track of the number of inversions between A and B which we have removed so far. When we are finished, there will be no inversions, so $count_{between}$ will be the number of inversions which we started with between A and B.

Remark 4.4. *A key advantage this algorithm has over the naïve algorithm is that it can count many inversions at once. This allows the algorithm to run in time $O(nlogn)$ rather than $O(n^2)$.*

5 Closest Pair of Points

Problem 5.1. *The closest pair of points problem is as follows. Given a set $P = \{p_1, \ldots, p_n\} \in \mathbb{R}^2$ of n points in the plane, which two points $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$ in P are closest to each other?*

Naïvely, solving this problem would take $O(n^2)$ time as we may have to consider every pair of points p_i, p_j. Using the following divide and conquer algorithm discovered by Shamos and Hoey, this problem can be solved in $O(nlogn)$ time.

Algorithm 5.2. *Given arrays X and Y of the points sorted by their x-coordinate and y-coordinate respectively, the algorithm does the following:*

1. *Base case: If $n \leq 1$ then the algorithm returns ∞ as there is no pair of points at all. If $n = 2$ then the algorithm returns p_1, p_2 as this is the only pair of points.*

2. *Dividing: We take $p_{i_{mid}} = X[\lfloor \frac{n}{2} \rfloor]$ to be the middle entry of X. We then take $P_L = \{p_i : i \text{ comes before } p_{i_{mid}} \text{ in } X\} \cup \{p_{i_{mid}}\}$ to be the points which come before $p_{i_{mid}}$ in X plus $p_{i_{mid}}$ itself and we take $P_R = \{p_i : i \text{ comes after } p_{i_{mid}} \text{ in } X\}$ to be the points which come after $p_{i_{mid}}$ in X.*

   After doing this, we take X_L and Y_L to be arrays containing the points in P_L sorted by their x-coordinate and y-coordinate respectively. Similarly, we we take X_R and Y_R to be arrays containing the points in P_R sorted by their x-coordinate and y-coordinate respectively. Constructing X_L and X_R can be done in $O(n)$ time by going through the array X annd putting the points in P_L and P_R into X_L and X_R respectively. Similarly, constructing Y_L and Y_R can be done in $O(n)$ time.

3. *Conquering: The algorithm recursively finds the closest pair of points p_{i_L}, p_{j_L} in the left half and the closest pair of points p_{i_R}, p_{j_R} in the right half.*

4. *Combining the solutions for each half into a full solution: To find the closest pair of points overall, we also need to check whether there are any pairs of points p_i, p_j in different halves which are closer than the pairs of points we've already found. We can do this as follows:*

   (a) *Letting $d = min\{d(p_{i_L}, p_{j_L}), d(p_{i_R}, p_{j_R})\}$, let $P_{center} = \{p_i = (x_i, y_i) : |x_i - x_{mid}| < d\}$ be the subset of points which are within distance d of the dividing line $x = x_{mid}$*

   (b) *Take Y_{mid} to be an array of the points in P_{mid} sorted by their y-coordinate.*

   (c) *Consider all pairs i, j such that one of the following is true:*

      i. *$i = i_L$ and $j = j_L$ or $i = i_R$ and $j = j_R$.*

      ii. *$p_i, p_j \in P_{mid}$ and p_j comes between 1 and 11 positions after p_i in Y_{mid}.*

   Take i_{min}, j_{min} to be the pair which minimizes $d(p_i, p_j)$ among all such pairs p_i, p_j and take $d_{min} = d(p_{i_{min}}, p_{j_{min}})$.

To show that this algorithm is correct, we need to confirm that any pair of points p_i, p_j on different sides which we did not check must be at least distance $d \geq d_{min}$ from each other. We first observe that

Proposition 5.3. *If $p_i \in P_L$ and $p_j \in P_R$ then $d(p_i, p_j) \geq d$ unless $p_i, p_j \in P_{mid}$*

Proof. Observe that $x_j - x_i = (x_j - x_{mid}) + (x_{mid} - x_i)$. If $p_i \notin P_{mid}$ then $x_j - x_i \geq x_{mid} - x_i \geq d$. If $p_j \notin P_{mid}$ then $x_j - x_i \geq x_j - x_{mid} \geq d$. $\square$

Now we just need to check pairs of points in P_{mid}. For this, the key idea is that the points in P_{mid} cannot be packed too densely.

Lemma 5.4. *For each point $p_i \in P_{mid}$, there are at most 11 points $p_j \in P_{mid}$ such that $0 \leq y_j - y_i \leq d$.*

Proof. This can be shown with the following very nice idea. We partition the region $M = \{(x, y) : |x - x_{mid}| \leq d\}$ into rows of 4 boxes where each box has side length $\frac{d}{2}$. Now observe that each such box can have at most 1 point in it because otherwise there would be a pair of points in either P_L or P_R which are less then distance d apart.

For any $p_i \in P_{mid}$, we can only have that $p_j \in P_{mid}$ and $0 \leq y_j - y_i \leq d$ if p_j is in the same row of boxes as p_i or is in the one of the two rows of boxes above p_i. Thus, there are only 11 boxes which p_j could be in so there can be at most 11 such points p_j. $\square$

Combining these observations, this algorithm checks all pairs of points which could possibly be within distance d of each other and is thus correct.

6 Karatsuba algorithm for multiplication

If we want to multiply two n-digit numbers, the grade school algorithm for this takes $O(n^2)$ time. Surprisingly, it is possible to multiply numbers much faster using a divide and conquer algorithm. The first such algorithm was the Karatsuba multiplication algorithm which works as follows.

Given two n-digit numbers x and y, taking $k = \lceil \frac{n}{2} \rceil$, we can write $x = 10^k x_1 + x_2$ and $y = 10^k y_1 + y_2$ where x_1 and y_1 are $(n-k)$-digit numbers and x_2 and y_2 are k-digit numbers. We now have that

$$xy = 10^{2k} x_1 y_1 + 10^k x_1 y_2 + 10^k x_2 y_1 + x_2 y_2$$

This allows us to compute xy by taking the sum of smaller products. Naïvely, we need to compute four products, which does not give us a faster algorithm. However, with some cleverness, we can instead only compute three products, $(x_1 + x_2)(y_1 + y_2)$, $x_1 y_1$, and $x_2 y_2$. The key observation is that $x_1 y_2 + x_2 y_1 = (x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2$. Using this observation, we have that

$$xy = 10^{2k} x_1 y_1 + 10^k \left((x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2\right) + x_2 y_2$$

The recurrence relation for the time taken by this algorithm is $T(n) = 3T(\frac{n}{2}) + O(n)$ which gives that $T(n)$ is $O(n^{log_2(3)})$.

Remark 6.1. *This algorithm works for any base. In particular, this algorithm also works for binary numbers.*

7 Advanced Topic: Convolutions and the Fast Fourier Transform

7.1 The Discrete Fourier Transform

Definition 7.1. *Given $N \in \mathbb{N}$ and a function $f : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$, the discrete Fourier Transform of f is the function $\hat{f} : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$ such that*

$$\hat{f}_k := \hat{f}(k) = \sum_{j=0}^{N-1} e^{\frac{-2\pi ijk}{N}} f(j)$$

The discrete Fourier transform has several nice properties, including the following:

Theorem 7.2.

1. $\forall k \in [0, N-1] \cap \mathbb{Z}, f(k) = \sum_{j=0}^{N-1} \frac{\hat{f}_j}{N} e^{\frac{2\pi ijk}{N}}$ *(Fourier decomposition)*
2. $N \sum_{j=0}^{N-1} |f(j)|^2 = \sum_{j=0}^{N-1} |\hat{f}_j|^2$ *(Parseval's Theorem)*

The fundamental reason these properties are true is that the Fourier characters form an orthonormal basis for the vector space of functions $f : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$. We make this statement more precise below.

Definition 7.3. *We define the kth Fourier character e_k to be the function $e_k : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$ such that $e_k(j) = \frac{1}{N} e^{\frac{2\pi ijk}{N}}$.*

Definition 7.4. *Given two functions $f, g : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$, we define the inner product of f and g to be*

$$\langle f, g \rangle = N \sum_{j=0}^{N-1} f(j)\bar{g}(j)$$

where $\bar{g}$ is the complex conjugate of g.

Remark 7.5. *The normalization for the inner product is strange but it is chosen to match the normalization for the discrete Fourier transform.*

Proposition 7.6. *The Fourier characters $\{e_k(j) : k \in [0, N-1] \cap \mathbb{Z}\}$ form an orthonormal basis for the vector space of functions $f : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$.*

Proof. Observe that

$$\forall k_1, k_2 \in [0, N-1] \cap \mathbb{Z}, \langle e_{k_1}, e_{k_2} \rangle = N \sum_{j=0}^{N-1} \frac{1}{N^2} e^{\frac{2\pi ij(k_1-k_2)}{N}}$$

which is 1 if $k_1 = k_2$ and is 0 otherwise. Thus, the Fourier characters $\{e_k(j) : k \in [0, N-1] \cap \mathbb{Z}\}$ are orthonormal. Since there are N Fourier characters and the vector space of functions $f : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$ has dimension N, the Fourier characters must form a basis as needed. $\square$

We can now prove Theorem 7.2.

Proof of Theorem 7.2. To see the first statement of Theorem 7.2, observe that if $f = \sum_{j=0}^{N-1} c_j e_j$ then $\hat{f}_k = \langle f, e_k \rangle = \langle \sum_{j=0}^{N-1} c_j e_j, e_k \rangle = c_k$. Thus, for all $j \in [0, N-1] \cap \mathbb{Z}$, $c_j = \hat{f}_j$ so $f = \sum_{j=0}^{N-1} \hat{f}_j e_j$ and thus $\forall k \in [0, N-1] \cap \mathbb{Z}, f(k) = \sum_{j=0}^{N-1} \frac{\hat{f}_j}{N} e^{\frac{2\pi ijk}{N}}$.

To see the second statement of Theorem 7.2, observe that since $f = \sum_{j=0}^{N-1} \hat{f}_j e_j$,

$$N \sum_{j=0}^{N-1} |f(j)|^2 = \langle f, f \rangle = \langle \sum_{j=0}^{N-1} \hat{f}_j e_j, \sum_{k=0}^{N-1} \hat{f}_k e_k \rangle = \sum_{j=0}^{N-1} \sum_{j=0}^{N-1} \hat{f}_j \overline{\hat{f}_k} \langle e_j, e_k \rangle = \sum_{j=0}^{N-1} |\hat{f}_j|^2$$

□

7.2 The Fast Fourier Transform

How fast can we compute the discrete Fourier transform $\hat{f}$ of a function $f : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$? If we do it naively, it will take $\Omega(n^2)$ time. However, if N is a power of 2, we can compute $\hat{f}$ in time $O(nlogn)$ using a divide an conquer algorithm, the fast Fourier transform. The algorithm works as follows:

Algorithm 7.7. *Given a function $f : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$ where N is a power of 2,*

1. *Take $f_{even} : [0, \frac{N}{2} - 1] \cap \mathbb{Z} \to \mathbb{C}$ to be the function such that $f_{even}(k) = f(2k)$ and take $f_{odd} : [0, \frac{N}{2} - 1] \cap \mathbb{Z} \to \mathbb{C}$ to be the function such that $f_{odd}(k) = f(2k+1)$*

2. *Compute the discrete Fourier transforms of $\hat{f}_{even}$ and $\hat{f}_{odd}$.*

3. *Observe that*

   (a) For all $k \in [0, \frac{N}{2} - 1] \cap \mathbb{Z}$,

$$\begin{aligned}
\hat{f}_k &= \sum_{j=0}^{N-1} e^{\frac{-2\pi ikj}{N}} f(j) \\
&= \sum_{j=0}^{\frac{N}{2}-1} e^{\frac{-2\pi ik(2j)}{N}} f(2j) + \sum_{j=0}^{\frac{N}{2}-1} e^{\frac{-2\pi ik(2j+1)}{N}} f(2j+1) \\
&= \sum_{j=0}^{\frac{N}{2}-1} e^{\frac{-2\pi ikj}{\frac{N}{2}}} f_{even}(j) + e^{\frac{-2\pi ik}{N}} \sum_{j=0}^{\frac{N}{2}-1} e^{\frac{-2\pi ikj}{\frac{N}{2}}} f_{odd}(j) \\
&= \hat{f}_{even}(k) + e^{\frac{-2\pi ik}{N}} \hat{f}_{odd}(k)
\end{aligned}$$

(b) For all $k \in [0, \frac{N}{2}-1] \cap \mathbb{Z}$,

$$\begin{aligned}
\hat{f}_{\frac{N}{2}+k} &= \sum_{j=0}^{N-1} e^{\frac{-2\pi i(\frac{N}{2}+k)j}{N}} f(j) \\
&= \sum_{j=0}^{\frac{N}{2}-1} e^{\frac{-2\pi i(\frac{N}{2}+k)(2j)}{N}} f(2j) + \sum_{j=0}^{\frac{N}{2}-1} e^{\frac{-2\pi i(\frac{N}{2}+k)(2j+1)}{N}} f(2j+1) \\
&= \sum_{j=0}^{\frac{N}{2}-1} e^{\frac{-2\pi ikj}{\frac{N}{2}}} f_{even}(j) + e^{\frac{-2\pi i(\frac{N}{2}+k)}{N}} \sum_{j=0}^{\frac{N}{2}-1} e^{\frac{-2\pi ikj}{\frac{N}{2}}} f_{odd}(j) \\
&= \hat{f}_{even}(k) - e^{\frac{-2\pi ik}{N}} \hat{f}_{odd}(k)
\end{aligned}$$

7.3 Convolutions

Definition 7.8. *Given two functions $f, g : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$, we define the convolution $f * g : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$ to be the function such that $(f * g)(k) = \sum_{j_1, j_2 : j_1 + j_2 = k \text{ mod } N} f(j_1)g(j_2)$*

Theorem 7.9. *For any two functions $f, g : [0, N-1] \cap \mathbb{Z} \to \mathbb{C}$ and any $k \in [0, N-1] \cap \mathbb{Z}$, $(f \hat{*} g)_k = \hat{f}_k \hat{g}_k$.*

Proof.

$$\begin{aligned}
(f \hat{*} g)_k &= \sum_{j=0}^{N-1} e^{\frac{-2\pi ikj}{N}} (f * g)(j) \\
&= \sum_{j=0}^{N-1} \sum_{j'=0}^{N-1} e^{\frac{-2\pi ik(j' + (j - j'))}{N}} f(j')g((j - j') \bmod N) \\
&= \sum_{j=0}^{N-1} \sum_{j'=0}^{N-1} e^{\frac{-2\pi ik(j'+j)}{N}} f(j')g(j) \\
&= \left(\sum_{j'=0}^{N-1} e^{\frac{-2\pi ikj'}{N}} f(j') \right) \left(\sum_{j=0}^{N-1} e^{\frac{-2\pi ikj}{N}} g(j) \right) = \hat{f}_k \hat{g}_k
\end{aligned}$$

□

Thus, if N is a power of 2 we can evaluate the convolution $f * g$ on all inputs in $O(NlogN)$ time by finding the discrete Fourier transforms $\hat{f}$ and $\hat{g}$ of f and g, taking $(f \hat{*} g) = \hat{f}\hat{g}$, and then inverting the discrete Fourier transform on $(f \hat{*} g)$ to find $f * g$.