Module 1 and Module 2 - Notes

Alperen Onur (z5161138)

August 2024

1 Gale-Shapely Algorithm

This algorithm solves the perfect stable matching problem of pairing n items with m other items such that no item belongs to two or more pairs (this is a simple matching) and all of n is matched with all of m. A perfect stable matching has the following condition on any two pairs (n_1, m_1) and (n_2, m_2) such that the following is never true

- n_1 prefers m_2 to m_1 **and**
- m_2 prefers n_1 to n_2

Algorithm preparation.

- Produce pairs in stages, with possible revisions.
- n items will attempt to pair with m items which are arbitrarily chosen. The m items will decide whether to accept the matching or not.
- An n item who is not paired is referred to as solo.
- Begin with all n as solo.

Algorithm execution

- While there is a solo n that has not attempted to pair with m, pick any solo n_i.
- n_i then attempts to pair with their highest preferenced m_j.
- If m_j is not yet paired, accept the matching tentatively.
- If m_j is already paired but prefers n_i over its current matching, pair m_j and n_i.
- If m_j is already paired but prefers its current pair over n_j, keep the current matching and decline the n_j matching.

This algorithm terminates in at most n^2 amount of matching attempts since every n can make at most n matching attempts.

2 Binary Search

An algorithm where you want to find some element in a **monotonic** sorted array.

Algorithm.

- **Divide:** Test the midpoint of the search range.
- **Conquer:** Search one side of the midpoint recursively. This is determined by your algorithm heuristic.
- **Combine:** Pass the answer up the recursion tree. Once the element is found, return it recursively.

The recursion is $\log_2 n$ levels deep with a total of $O(1)$ time spent at each level. $O(\log n)$ time complexity.

3 Unbounded Binary Search

To apply binary search, you need a well-defined interval. If the problem doesn't contain an interval to search over, then we need to define one.

Let A be a sorted array of distinct elements where the length of A is unknown. Find the smallest index i such that $A[i] > 0$ or report no such index exists.

Algorithm.
We define the search s[ace as each power of 2 element. That is, checking elements at $1, 2, 4, 8, 16, ..., 2^k$. For each element, if $A[2^k] > 0$ then we can search $A[2^{k-1}, .., 2^k]$. This is the interval we will binary search over. This search will take $O(k)$ time which is logarithmic time since $1, 2, 4, 8, 16, ...$ increments exponentially.

4 Maximum Median

Given a sorted array A, we want to know whether we can make the median of the numbers in the array at least some value t where we can increase the values k number of times.

Algorithm.
Let n be the index of the median of the array. We iterate over indices i from n to $2n - 1$ to calculate

$$\sum_{i=n}^{2n-1} max(t - A[i], 0)$$

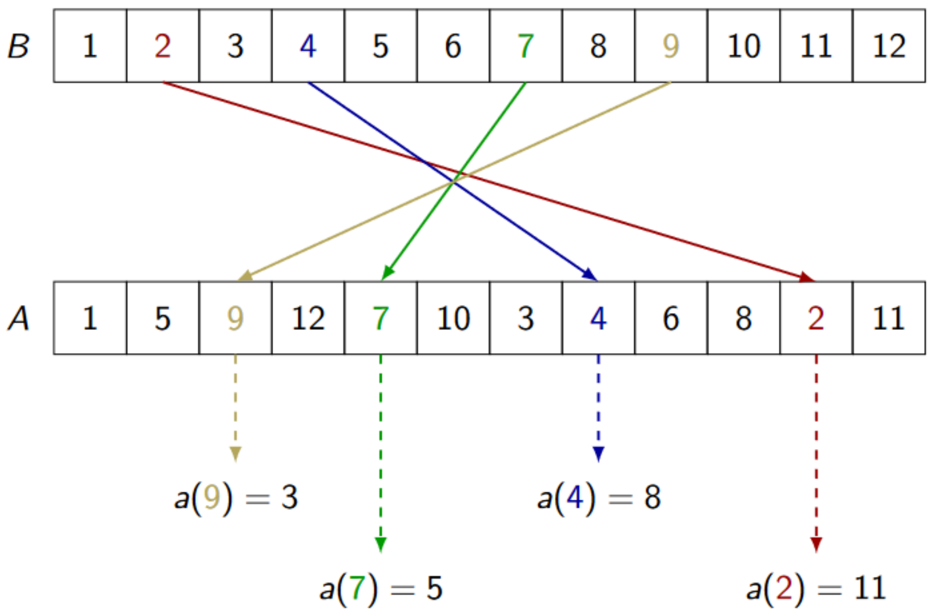
If this value is $\leq k$, report yes, otherwise report no. Algorithm runs in $O(n)$ time.

Monotonic.
If target t in the previous problem is achievable, then clearly any smaller target $t - l$ is also achievable and if target t is not achievable then any target $t + l$ is also not achievable. Targets are achievable until some cutoff and impossible thereafter, this is the definition of monotonic. Compute these t values as they're needed with a binary search (imagine there's a separate array of k length with t values that you're binary searching). The search is performed in $\log k$ steps and each step taking $O(n)$ time and so the time complexity is $O(n \log k)$.

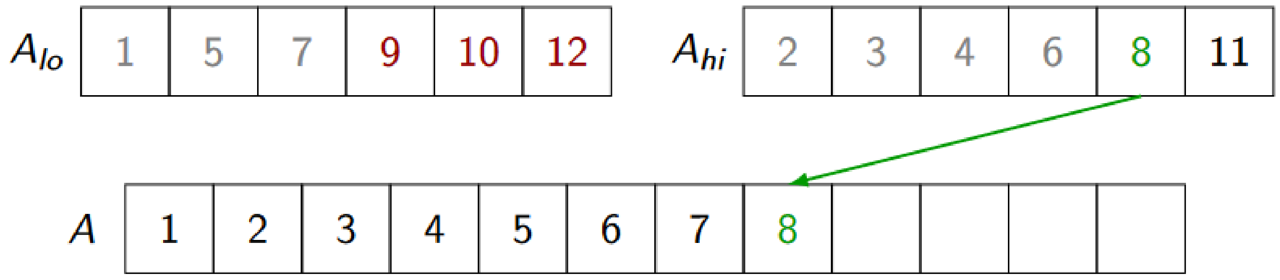
5 Inversion counting.

An inversion is a pair (i, j) in an array such that:

- $i < j$ (these refer to the index numbers) and
- $a(i) > a(j)$ (the values at those indices)



A brute force approach would take $O(n^2)$ time but we can do better; $O(n \log n)$. We can do this by tweaking the merge sort algorithm to count these inversions whilst it sorts the array. In an array A where $I(A)$ determines the inversions in A in the combine step of each merge of partitions A_{lo} and A_{hi}. For each time an element in A_{hi} is chosen in the sort, we have an inversion between that element in A_{hi} and all the remaining elements in A_{lo}. Calculate all these inversions recursively at every combine step of the merge sort and the sum of all the inversions is the total inversions in the array.



$$\text{count} = 5 + 5 + 5 + 4 + 3 + 1 = 23.$$

6 Divide and Conquer.

There should be 4 essentials to consider at every stage of a divide and conquer algorithm.

- **Divide:** split up a large instance into smaller instances of the same type.
- **Conquer:** apply the same algorithm to recursively solve each small subproblem, independently of each other.
- **Combine:** synthesis the solution to the original instance from the subproblem solutions.
- **Base case(s):** the smallest instances, where no further recursion is possible. Usually trivial.

Correctness.
The idea is that if the base cases are solved correctly, and the combine step is correct, then the subproblems of all sizes are solved correctly, by fast induction. A more formal proof is as follows:

- **(Prove $k = 1$)** The base cases are typically easy to argue because, by the nature of how you have handled the base cases, the problem is solved.
- **Assume $k = n$** Assume now, that the problem has been solved for each recursive steps.
- **(Prove $k = 2n$ or however many divisions)** We now need to argue that if our algorithm retrieves the correct solution in our recursive stages, then it also retrieves the correct solution in the current step.

7 Identity Element.

Let $A[1..n]$ be a sorted array of n distinct integers where $A[1] > 1$. Some of these integers may be positive, negative or zero. Design an $O(1)$ algorithm to decide if there exists some index i such that $A[i] = i$.

Algorithm.
The solution for this algorithm expects you to construct an array $B[i] = A[i] - i$. If $B[i] = 0$ then you return YES, otherwise return NO.

8 Search in Rotated Sorted Array

Given a sorted array A that has been shifted to the right be k units, design an $O(\log n)$ algorithm that decides if x appears somewhere in A.

Algorithm.
If we know what k is then we can just binary search both sides of the pivot k. We can binary search to find k. We know that for each half of the array, one half will be sorted and the other half will not be sorted. So binary search on the half that isn't sorted which can be checked by comparing the value your binary search i with the edges of the array $A[1]$ and $A[n]$. If $i < A[n]$, search left of i. If $i > A[1]$, search right of i.

9 Order Statistic of Two Sorted Arrays

Let $A[1..n]$ and $B[1..n]$ be two sorted arrays. For simplicity, assume that A and B share no common elements and all elements in each array are distinct. Construct a new array $C[1..2n]$ by merging A and B together.

Algorithm.

- If $k < n$, then the kth element must be the median of $A[1..k] \bigcup B[1..k]$
- If $n \leq k \leq 2n$, then the kth smallest element must be the median of $A[(k-n)..n] \bigcup B[(k-n)]..n$. Since there are $n-(k-n)+1 = 2n-k = 1$ elements in each array, there are $2(2n-k)+1$ elements altogether. Therefore, x must be larger than $2(k-n)-1$ elements since there are larger than all elements in $A[(k-n)..] \bigcup B[(k-n)..n]$. But these elements are larger than $2((k-n)-1)$ elements since they are larger than all elements in $A[1..(k-n-1)] \bigcup B[1..(k-n-1)]$. Therefore, x is larger than $(2n-k)+1+2(k-n)-2 = k-1$ elements. Thus, x is the kth smallest element.
- If $A[\frac{n}{2}] > B[\frac{n}{2}]$, then firstly $A[\frac{n}{2}]$ is larger than all $\frac{n}{2}$ elements in the subarray $A[1..(\frac{n}{2}-1)]$. But since $A[\frac{n}{2}] > B[\frac{n}{2}]$, then it follows that $A[\frac{n}{2}] > B[j]$ for all $j < \frac{n}{2}$. Therefore, $A[\frac{n}{2}]$ must be larger than all $2(\frac{n}{2})-1 = n-1$ elements in the array $A[1..(\frac{n}{2})-1] \bigcup B[1..\frac{n}{2}]$ and so, must be at least as large as the median. But this implies that all of the elements that are larger than $A[\frac{n}{2}]$ must necessarily be larger than the median and so, we can safely remove all $\frac{n}{2}$ elements in the subarray $A[(\frac{n}{2}+1)..n]$.

  By similar reasoning, we can safely remove all $\frac{n}{2}$ elements in the subarray $B[1..\frac{n}{2}]$ since $B[i] < B[\frac{n}{2}] < A[\frac{n}{2}]$ is smaller than the median.
- The argument is symmetric in the case where $A[\frac{n}{2}] < B[\frac{n}{2}]$. In other words we can safely remove the $\frac{n}{2}$ elements in the subarray $B[(\frac{n}{2}+1)..n]$ and all $\frac{n}{2}$ elements in the subaray $A[1..\frac{n}{2}]$

10 Recurrences and the Master Theorem.

We can form an equation representing the growth rate of a recurrence in a Divide and Conquer algorithm. From this equation, we can obtain a time complexity for our algorithm.
Let $a \geq 1$ be an integer and $b > 1$ be a real number and suppose that a divide and conquer algorithm:

- reduces a problem of size n to a many subproblems of size $\frac{n}{b}$
- with overhead cost of $f(n)$ to split up the problem and combine the solutions from these smaller subproblems.

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

As we go down the recursion tree, the number of subproblems grows exponentially but the size of those subproblems shrink exponentially. We'll use a critical exponent $c^* = \log_b a$ and critical polynomial n^{c^*} to separate the Master Theorem into 3 cases.

Master Theorem cases.

Case 1) Branching by a factor of a outweighs the work done in the combine step $\Rightarrow$ all the work is done at the leaves. If $f(n) = O(n^{c^*-\epsilon})$ for some $\epsilon > 0$, then $T(n) = O(n^{c^*})$.

Case 2) The effects of the branching is roughly equivalent to the amount of work done in the combine step $\Rightarrow$ each level requires about the same of work. If $f(n) = O(n^{c^*})$, then $T(n) = O(n^{c^*} \log n)$.

Case 3) Branching is outweighed by the reduction in the combine step $\Rightarrow$ all the work is done at the root. If $f(n) = \Omega(n^{c^*+\epsilon})$ for some $\epsilon > 0$, $k < 1$ and some $n_0 = af(\frac{n}{b}) \leq kf(n)$ holds for all $n > n_0$(regularity condition), then $T(n) = O(f(n))$.

11 The Karatsuba trick

Say we have two numbers A and B, each represented by n number of bits were A_0, B_0 represents the least significant $\frac{n}{2}$ bits and A_1, B_1 represents the most significant $\frac{n}{2}$ bits. Then $A = A_1 2^{\frac{n}{2}} + A_0$ and $B = B_1 2^{\frac{n}{2}} + B_0$. We can calculate the product in polynomial time recursively using the following equation

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0.$$

The Karatsuba trick can do better, specifically $O(n^{\log_2 3})$. We can rearrange the above equation to save one multiplication per recursion

$$AB = A_1 B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0) 2^{\frac{n}{2}} + A_0 B_0.$$

Algorithm.
Recursively apply the algorithm to multiply two integers A and B, each consisting of n bits. The base case is $n = 1$, where we simply evaluate the product. Otherwise, compute the following recursively.

- $X = A_0 B_0$
- $W = A_1 B_1$
- $V = (A_1 + A_0)(B_1 + B_0)$

Finally, compute $W 2^n + (V - W - X) 2^{\frac{n}{2}} + X$ by summing

- W shifted left by n bits
- V less W and X shifted left be $\frac{n}{2}$ bits, and
- X with no shift.

Time Complexity.
$T(n) = 3T(\frac{n}{2}) + cn$ where $a = 3$ and $b = \frac{n}{2}$ from the 3 multiplications of $\frac{n}{2}$ numbers. This falls into case 1 of the Master Theorem but $T(n) = O(n^{\log_2 3}) \neq \Omega(n^2)$

12 Convolutions and the Fast Fourier Transform

Polynomials can be represented as coefficients in the following form

$$P(x) = A_n x^n + A_{n-1} x^{n-1} + .. + A_1 x A_0$$

We can use Horner's Rule to evaluate the polynomial in $O(n)$ time

$$P(x) = A_0 + x(A_1 + A_2 x + ... + A_{n-1} x^{n-2} + A_n x^{n-1})$$
$$P(x) = A_0 + x(A_1 + x(A_2 + x(...)))$$

The issue arises however when we want to multiply 2 polynomials, this can only be done in $O(n^2)$ using the coefficient representation.

Convolution.
In general, each coefficient of $P_c(x)$ is given by

$$C_t = \textstyle\sum_{i+j=t} A_i B_j$$

Directly computing the product of two polynomials is equivalent to working out each term of the convolution using the above formula as both involve multiplying each A_i by each B_j. This takes $O(n^2)$ time however as there are quadratic many $A_i B_j$ products. However we can do better, $O(n \log n)$ time, by computing the convolution indirectly where n represents the degree of the largest input polynomial. This is done by converting the polynomial to a its value representation in $O(n \log n)$ time and calculating the product in $O(n)$ time.

Value Representation.
Addition and multiplication can be executed in pointwise linear time

- $(P_A + P_B)(x_0) = P_A(x_0) + P_B(x_0)$
- $(P_A P_B)(x_0) = P_A(x_0) P_B(x_0)$

Fast Fourier Transform.
The Discrete Fourier Transform evaluates these polynomials $P_A(x)$ and $P_B(x)$ at $2n+1$ distinct points $x_0, x_1, .., x_{2n}$. Then we multiply them point by point using $2n+1$ multiplications of large numbers to get $2n+1$ values of the product polynomial $P_C(x)$. Finally we reconstruct the coefficients of $P_C(x)$ like so

$$P_C(x) = C_{2n}x^{2n} + C_{2n-1}x^{2n-1} + ... + C_1x + C_0.$$

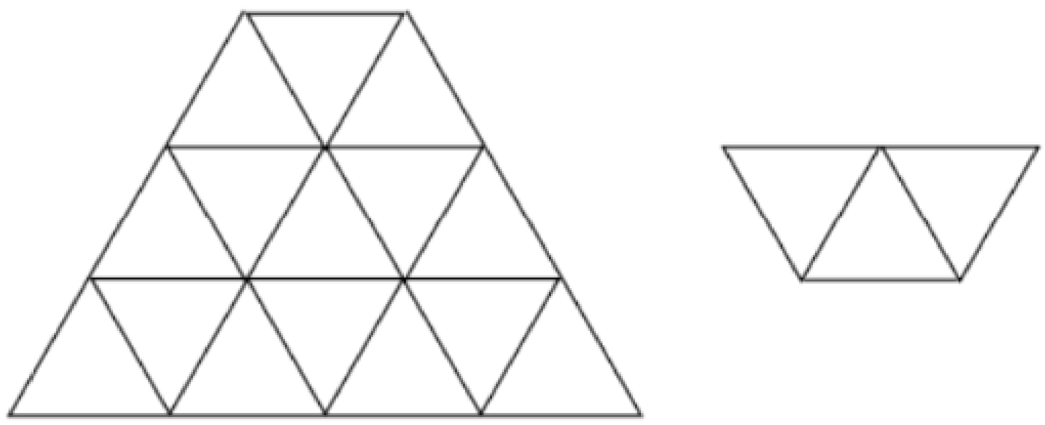
The Fast Fourier Transform instead does the above by using divide and conquer to split the polynomials $P_A(x)$ and $P_B(x)$ into even and odd powers. Let $A^{[0]} = \{A_0, A_2, A_4, .., A_{m-2}\}$ and $A^{[0]} = \{A_1, A_3, A_5, .., A_{m-1}\}$ such that each of these polynomials have $\frac{m}{2}$ coefficients each and $P_A(x)$ has m coefficients. The details after this step are beyond the scope for the final exam but essentially, this divide allows us to reduce the problems to size $\frac{m}{2}$ plus some linear overhead.

$$T(m) = 2T(\tfrac{m}{2}) + cm$$

This falls into case 2 of the Master Theorem and so $T(m) = O(m \log m) = O(n \log n)$.

13 Tiling Problems

Let n be a power of two. An equilateral triangle is partitioned into smaller equilateral triangles by parallel lines dividing each of its sides into $n > 1$ equal segments. The topmost equilateral triangle is cut off. We want to tile the remaining equilateral triangles with trapezoids, each of which is composed of three equilateral triangles.



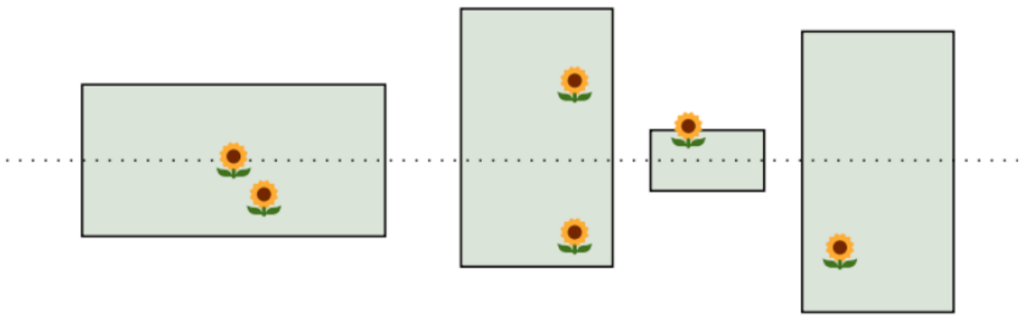
Design a divide and conquer algorithm to tile the equilateral triangles with trapezoids. Justify the correctness of the algorithm and analyse its time complexity.

Algorithm.

- Base case: A triangle consisting of 4 equilateral triangles. Place a trapezium in 3 of the equilateral triangles. 1 equilateral triangle will always be left empty.
- Divide: Reduce the problem by dividing the board into 4 smaller equilateral triangles.
- Conquer: Place trapezium such that it causes 3 other triangles to also have a missing piece.
- Combine: Combine all trapezium placements, including filling missing pieces with an extra trapezoid, to obtain original triangle being filled with trapeziums.
- Time Complexity: $T(n) = 4T(\frac{n}{2}) + O(1) = O(n^2)$.

14 Flowers in Garden

Alice is planing n_1 flowers $f_1, ..., f_{n_1}$ among n_2 rectangular gardens $g_1, ..., g_{n2}$. Bob's task is to determine which flowers belong to which gardens. Alice informs Bob that no two gardens overlap; therefore, if a flower belongs to a garden, then it belongs to exactly one garden. Moreover, a garden can contain multiple flowers. If a flower does not belong to any garden, then Bob returns undefined for that flower. Finally, let $n = n_1 + n_2$.



We can define the location of a rectangular garden by identifying its bottom-left and top-right corners. Additionally, flower f_i is represented by a point $F[i]$. Formally, we are given three arrays:

- $B = [(x_1, y_1), ..., (x_{n_2}, y_{n_2})]$, where $B[i]$ represents the bottom-left point of garden g_i.
- $T = [(x_1, y_1), .., (x_{n_2}, y_{n_2})]$, where $T[i]$ represents the top-right point of garden g_i.
- $F = [(x_1, y_1), .., (x_{n_1}, y_{n_1})]$, where $F[i]$ represents the location of flower f_i.

For each flower f_i, your task is to identify which garden (if any) contains f_i. If a flower is not contained inside any garden, then we return undefined.

Algorithm.
Assume all the gardens intersect with a horizontal line. We first begin to solve this problem by sorting all arrays by their x coordinate. Now, simply sweeping across the gardens array to see what flowers belong to that respective garden. Essentially, you're maintaining 3 points between all these arrays. We can do all the aforementioned in $O(n \log n)$ time using merge sort

Now, remove the assumption of the gardens intersecting with a single horizontal line. We can achieve a $T(n) = 2T(\frac{n}{2}) + O(n \log^2 n)$ time complexity by using our previous solution where the gardens intersected one line. Begin by sorting T, B and F by their y coordinate. Assume that you are combining flowers and gardens into an array of size n so that we can use the single horizontal line approach.

- Divide: Split by the median Y coordinate.
- Conquer: Repeat for the top and bottom halves.
- Combine: All that intersect using part A.
- Base Case: When array is size $n \leq 1$. This will either return 1 flower, 1 garden, or nothing.

15 Year of the Dragon

It is the Year of the Dragon, and you just so happen to have a collection of n coloured dragon figurines. Each figurine is coloured on of c colours. Your friend wants to keep a set of k dragon figurines from your collection: Two sets of dragon figurines are considered the same if they contain the same number of figurines of each colour.

Design an $O(ck \log k)$ algorithm that returns the number of different ways of choosing k dragon figurines. Your input is an array $Dragon[1..c]$ and an integer k, where $Dragon[i]$ denotes the number of dragon figurines of colour i. Your output is a non-negative integer. You may assume that $k \leq n$.

Algorithm.
We can define the $Dragon[j]$ as the number of figurines of colour j. So $P_j(x) = 1 + x + x^2 + .. + x^{Dragon[j]}$.

Essentially, this polynomial has terms for each power of x up to the number of figures we have - so each possible number of figures we can take is a term in the polynomial. Think of x as representing different figures in our collection. This considers every possible number of figurines of colour j that you can pick.

$$P_j(x) = \sum_{k=0}^{Dragon[j]} x^k$$

Then we can combine this for every colour by multiplying all polynomials of different dragon figurine colours together. $P(x) = a + bx + cx^2 + ... + zx^n$.

$$P(x) = \Pi_{j=1}^{c} P_j(x)$$

There are c multiplications each taking $O(n \log n)$ tine. We can me it $O(k \log k)$ time by discarding terms greater than x^k since we can choose a maximum of k toys. Therefore our final complexity is $O(ck \log k)$.