

Problem 1 - Finding the Majority Element

An immediate observation is that, if n is odd, e is the majority element of $A[1..n]$ and $A[n] \neq e$, then e is also the majority element of $A[1..n-1]$.

Suppose, now, that n is even. Let us consider the following procedure: let us pair up element $A[2i-1]$ with element $A[2i]$ of the array, for all $i \in \{1, \dots, n/2\}$; then, for each pair, if the two elements of the pair are equal, let us keep one of the elements and, if they are not, discard both of them. Call B the set of elements that survived this procedure, where $|B| \leq n/2$. We claim the following.

Claim 1 *If n is even, e is the majority element of $A[1..n]$ and B is the set of elements which survived the above procedure, then B has a majority element which is equal to e .*

Proof: Suppose that k is the number of pairs created by the above procedure, in which both elements are equal to e . Suppose, further, that ℓ is the number of pairs created by the procedure which contain unequal elements. Clearly, $|B| = n/2 - \ell$. Moreover, since e appears in A at least $n/2 + 1$ times it must hold that $2k + \ell \geq n/2 + 1$. This implies

$$k \geq \frac{n/2 - \ell}{2} + \frac{1}{2} \Rightarrow k \geq \frac{|B|}{2} + \frac{1}{2}.$$

Hence, e is a majority element of B . ■

It is not hard to design an algorithm for finding a majority element based on the above. Algorithm FINDCANDIDATE described below finds a candidate for the majority element. Then a pass over the array, using TESTMAJORITY verifies if the candidate element is indeed a majority element. It is clear from the above discussion that, if A has a majority element, then FINDCANDIDATE(A) will return that element. Moreover, since we always verify whether the candidate element is indeed the majority element we will never output a non-majority element.

Algorithm FINDCANDIDATE($A[1..n]$)

If n **is odd then**

If TESTMAJORITY($A[1..n]$, n) = **true**

return $A[n]$;

Else

$n := n - 1$;

$B := []$ /* an empty array */; $\ell := 1$;

For $j := 1$ **to** $n/2$

If $A[2j-1] = A[2j]$ **then** $B[\ell] := A[2j]$; $\ell := \ell + 1$;

If B **is empty**

return null;

Else

return FINDCANDIDATE($B[1\dots\ell - 1]$)

where

Algorithm TESTMAJORITY($A[1\dots n]$, i)

counter := 0;

for $j := 1$ **to** n

If $A[j] = A[i]$ **counter** := **counter** + 1;

if **counter** > $n/2$ **return** TRUE;

The running time $T(n)$ of the procedure FINDCANDIDATE on an array of size n satisfies

$$T(n) = T(\lfloor n/2 \rfloor) + O(n),$$

where the $O(n)$ accounts for the call of TESTMAJORITY from within FINDCANDIDATE in the case n is odd and $T(\lfloor n/2 \rfloor)$ accounts for the recursive call of FINDCANDIDATE on an array of size at most $n/2$. From Master Theorem, it follows that $T(n) = O(n)$.

Since the extra call to TESTMAJORITY needed for verifying whether the candidate element is indeed a majority element takes linear time we need linear time overall.

Problem 2 - Fast Fourier Transform

(a) By inspection $\omega = 3$ satisfies the required property. Indeed,

$$(\omega, \omega^2, \omega^3, \omega^4, \omega^5, \omega^6) = (3, 2, 6, 4, 5, 1) \pmod{7}$$

and

$$\sum_{i=1}^6 \omega^i = 21 = 0 \pmod{7}.$$

(b) The matrix of the Fourier transform is $M_6 = [\omega^{ij}]_{0 \leq i, j \leq 5}$, i.e.

$$M_6 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{pmatrix}.$$

Hence the Fourier transform of the sequence $v = (0, 1, 1, 1, 5, 2)$ modulo 7 is

$$M_6 v = \begin{pmatrix} 10 \\ 41 \\ 25 \\ 30 \\ 31 \\ 31 \end{pmatrix} \pmod{7} = \begin{pmatrix} 3 \\ 6 \\ 4 \\ 2 \\ 3 \\ 3 \end{pmatrix}.$$

(c) In modulo 7 arithmetic, the inverse of $\omega = 3$ is $\omega^{-1} = 5$, since $3 \cdot 5 = 1 \pmod{7}$. We will argue that if we define

$$M_6^{-1} = N^{-1} \cdot [(\omega^{-1})^{ij}]_{0 \leq i, j \leq N-1} \pmod{7},$$

where $N = 6$, then

$$M_6 \cdot M_6^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \pmod{7},$$

i.e. the product of the two matrices modulo 7 gives the identity matrix. Indeed, let us consider the inner product of row i of matrix M_6 with column j of matrix M_6^{-1} . The inner product is equal to

$$\sum_{k=1}^N \omega^{ik} \cdot \omega^{-kj} = \sum_{k=1}^N (\omega^{i-j})^k.$$

If $i = j$, the above expression is equal to N since $\omega^{i-j} = 1$.

If $i \neq j$, we claim that the above expression equals 0. Indeed, recall that the following is true for every x

$$x^N - 1 = (x - 1) \cdot \sum_{k=0}^{N-1} x^k$$

Hence, it is also true modulo 7

$$x^N - 1 = (x - 1) \cdot \sum_{k=0}^{N-1} x^k \pmod{7}.$$

In particular, it is true for $x = \omega^{i-j}$ which implies

$$(\omega^N)^{i-j} - 1 = (\omega^{i-j} - 1) \cdot \sum_{k=0}^{N-1} (\omega^{i-j})^k \pmod{7}.$$

But, since $N = 6$, $\omega^N = 1 \pmod{7}$. Moreover, since $i, j \in \{0, \dots, 5\}$ and $i \neq j$, it follows that $i - j \in \{-5, \dots, -1, 1, \dots, 5\}$. But for each of these values $\omega^{i-j} - 1 \neq 0 \pmod{7}$, hence $\omega^{i-j} - 1$ has an inverse. If we multiply both sides of the above equality by this inverse we get

$$\sum_{k=0}^{N-1} (\omega^{i-j})^k = 0 \pmod{7}.$$

Hence, our choice for M_6^{-1} is justified. The matrix is

$$M_6^{-1} = \begin{pmatrix} 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 2 & 3 & 1 & 5 & 4 \\ 6 & 3 & 5 & 6 & 3 & 5 \\ 6 & 1 & 6 & 1 & 6 & 1 \\ 6 & 5 & 3 & 6 & 5 & 3 \\ 6 & 4 & 5 & 1 & 3 & 2 \end{pmatrix}.$$

and if we multiply it by $(3, 6, 4, 2, 3, 3)^T$ we get our initial vector.

(d) Define the vectors

$$v_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \text{and} \quad v_2 = \begin{pmatrix} -1 \\ 2 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

corresponding to the given polynomials. The FFT's of v_1 and v_2 are

$$\eta_1 = M_6 v_1 = \begin{pmatrix} 3 \\ 6 \\ 0 \\ 1 \\ 0 \\ 3 \end{pmatrix} \quad \text{and} \quad \eta_2 = \begin{pmatrix} 2 \\ 4 \\ 4 \\ 3 \\ 1 \\ 1 \end{pmatrix}.$$

By coordinate-wise multiplication of these vectors, we find that the FFT η of the vector v corresponding to the product of the polynomials is

$$\eta \equiv M_6 v = \begin{pmatrix} 6 \\ 3 \\ 0 \\ 3 \\ 0 \\ 3 \end{pmatrix}$$

so that the vector corresponding to the product of the polynomials is

$$v = M_6^{-1} \eta = \begin{pmatrix} 6 \\ 1 \\ 1 \\ 3 \\ 1 \\ 1 \end{pmatrix}.$$

Hence,

$$(x^2 + x + 1)(x^3 + 2x - 1) = x^5 + x^4 + 3x^3 + x^2 + x + 6 \pmod{7}.$$

Problem 4 - Dijkstra's Algorithm

We will create a data structure which uses $O(W)$ space and results in $O(1)$ -time update operations and $O(W)$ -time extract min operations. Hence, using this data structure for the implementation of Dijkstra's algorithm results in running time of $O(W|V| + |E|)$. The data structure is based on the following observation.

Lemma 1 *If all edge weights in the graph are from the set $\{1, \dots, W\}$ then at every step t in the execution of Dijkstra's algorithm the following is satisfied*

$$\forall u, v \in V - S_t : (d(u) \neq \infty \wedge d(v) \neq \infty) \implies |d(u) - d(v)| \leq W,$$

where S_t is the set of elements for which the shortest path distance to the source has already been found at time t .

Proof: Since $u, v \in V - S_t$ and $d(u) \neq \infty$, $d(v) \neq \infty$, it follows that there exist some $x, y \in S_t$ such that

$$d(u) = d(x) + w(x, u)$$

and

$$d(v) = d(y) + w(y, v).$$

Moreover, since Dijkstra's algorithm picks nodes in increasing order of distances it must be that $\max(d(x), d(y)) \leq \min(d(u), d(v))$. Supposing, without loss of generality, that $d(u) \geq d(v)$ we have that

$$|d(u) - d(v)| = d(u) - d(v) = d(x) + w(x, u) - d(v) = w(x, u) - (d(v) - d(x)) \leq w(x, u) \leq W.$$

■

It follows that at every step of the algorithm the distances of the nodes in the set $V - S_t$ span a range of at most $W + 1$ consecutive values provided that they are not infinity. This suggests keeping the nodes of the set $\{v | v \in V - S_t, d(v) \neq \infty\}$ in a circular array $A[0..W]$ of length $W + 1$. Every cell of the array keeps a doubly linked list of nodes which have currently the same distance from the source and there is a head pointer $h \in \{0, \dots, W\}$ pointing to the cell of the array containing the nodes which currently have the smallest distance from the source. If d is that distance, then the nodes in cell i of the array are at current distance $d + \ell$ from the source where $\ell \in \{0, \dots, W\}$ is such that $h + \ell = i \pmod{W + 1}$. Of course, each node keeps on its own the actual distance from the source since the index of the cell where an element is placed will not reflect the actual distance from the source. Suppose finally that we keep the nodes v with $d(v) = \infty$ in some separate doubly linked list.

Let us consider the update and extract minimum operations on the data structure. The extract min works as follows: one element v is extracted in $O(1)$ time from the linked list contained at cell $A[h]$ of the array; if there is no such element, i.e. the list is empty, we return that the priority queue is empty altogether; if the operation succeeds but the list at $A[h]$ becomes empty, then the value of h must be updated. This is done by a circular scan of the array, starting at h and increasing modulo $W + 1$, which stops at the first cell of the array which is non-empty. If all cells of the array are found empty, we leave $h = 0$. This takes at most $O(W)$ time.

The update operation works as follows: the node which is updated, say node v , is removed from the doubly linked list where it is currently placed. If $h = 0$ and $A[0]$ is empty then v becomes the only node of the cell $A[0]$. Otherwise, the new distance d of v is compared to the distance d' of (one of) the nodes stored at the head of the circular array, i.e. position $A[h]$; if $j = d - d' \geq 0$ then v is placed in the linked list of the cell $A[h + j \pmod{W + 1}]$ of the array; if $j = d - d' < 0$ then v is placed at cell $A[h - j \pmod{W + 1}]$ of the array and the header is also updated $h := h - j \pmod{W + 1}$; it is actually not hard to see based on the above lemma that the cell where we placed v was empty in this case. All operations take $O(1)$ time.

Problem 5

The *intuition* comes by considering the amortized cost of the ZIG operation versus that of the ZIG-ZAG and ZIG-ZIG operations. We saw in class that the amortized cost of a ZIG operation on some element x is upper bounded by $1 + 2(r'(x) - r(x))$, where $(r'(x) - r(x))$ is the change in the

rank of the element after the operation; hence pulling a leaf of the tree to the root via a sequence of ZIG operations would require an amortized cost of at most

$$h(x) + 2(r''(x) - r(x)),$$

where $h(x)$ is the depth of x , i.e. the number of edges between x and the root of the tree, before starting moving x towards the root and $(r''(x) - r(x))$ is the overall change in rank. Since $h(x)$ can be as bad as $O(n)$ the bound becomes

$$O(n) + 2(r''(x) - r(x)),$$

for one splay operation, which is much worse than the bound of $O(\log n)$ which we derived for splaying using a sequence of ZIG-ZIGs, ZIG-ZAGs and potentially one ZIG operation in the end. Hence, if we imagine a sequence of m splay operations each of which splays a leaf at distance $O(n)$ from the root that would result in a bound of $O(m \cdot n)$ using only ZIGs, as opposed to the bound $O((m+n) \log n)$ that the same sequence of operations would require if we were also to use ZIG-ZIGs and ZIG-ZAGs.

However, to **formally** prove that the use of ZIG-ZIGs and ZIG-ZAGs outperforms the plain use of ZIGs we have to face the following flaws of the above argument

- what we did above is to compare upper bounds for a sequence of ZIG operations with upper bounds for the ZIG-ZIG and ZIG-ZAG operations; it could be the case that our upper bounds are very tight for the latter and very loose for the former, hence our reasoning might not reflect the reality;
- even so, our analysis was based on the fact that it is possible to construct a tree and a sequence of searches each of which splays a leaf at depth $O(n)$ from the root; however, ZIGs result in reshaping the structure of the tree they are performed on and we cannot trivially assume that the claimed sequence exists.

Hence, to be more formal we will show an example of a tree on n leaves and a sequence of $\Omega(n)$ searches on the tree such that, if splaying is done using ZIG operations, all nodes which are splayed are leaves at distance $\Omega(n)$ from the root. This sequence of searches will result then in $\Omega(n^2)$ time, whereas if we were to use ZIG-ZIGs and ZIG-ZAGs that would drop down to $O(n \log n)$. Our example tree is a chain of elements u_1, u_2, \dots, u_n , where u_1 is the root and the set of edges is $\{(u_i, u_{i+1}) | i \in \{1, 2, \dots, n-1\}\}$. Clearly, searching for u_n will take $\Omega(n)$ time since its distance from the root is $n-1$. We claim that after splaying element u_n , element u_{n-1} will be located at distance $n-2$ from the new root, which will be element u_n , so splaying that will take $\Omega(n)$ time as well. Let us continue by querying $u_{n-2}, \dots, u_{n/2}$. We claim that, for all $k \in \{n/2, \dots, n-2\}$, when we query for element u_k the element is located at distance $k-1$ from the root; hence the splay operation requires $\Omega(n)$ time for each of these elements. Assuming that these claims are true (which is left as an exercise), the time complexity for querying $u_n, u_{n-1}, \dots, u_{n/2}$ is $\Omega(n^2)$ as opposed to $O(n \log n)$.

Stable Matchings - Problem 1 Chapter 2 of Kleinberg and Tardos

The statement is true. To prove this, suppose, by way of contradiction, that there exists a stable matching S containing the pairs (m, w') and (m', w) where $m \neq m'$ and $w \neq w'$. Since m is higher in the list of w than m' and w is higher in the list of m than w' , there exists an instability in S . So it must be that (m, w) is contained in every stable matching.

BFS and DFS trees - Problem 3 Chapter 6 of Kleinberg and Tardos

Recall that the edges of a graph which are not present in a BFS tree of the graph are *cross edges*, i.e. edges (u, v) such that u is not an ancestor of v in the tree and v is not an ancestor of u . By assumption, since the BFS and DFS trees are the same, it must be that the same edges are missing from the DFS tree. However, in the case of undirected graphs, the only edges missing from the DFS tree are *backward edges*, i.e. edges (u, v) so that either u is an ancestor of v or v is an ancestor of u . Since the BFS can be missing only cross edges and the DFS tree only backward edges, it follows that, in order for the trees to be the same, none of them can be missing any edges of G . Hence, G must be a tree which is precisely equal to the BFS and DFS trees.

Kruskal's Algorithm - Problem 4 Chapter 11 of Kleinberg and Tardos

We will argue that for every MST T of a given graph G , there exists a valid ordering \mathcal{O} of the edges of G so that Kruskal's algorithm returns T under that ordering.

It will be useful to observe that the output of Kruskal's algorithm depends only on the input graph G and the order in which the edges are being considered; i.e. it does not depend on the precise weights of the edges per se except as a means to define the ordering.

Having made this observation let us proceed to construct a valid ordering \mathcal{O} corresponding to some MST $T = (V, E)$ of the given graph G . Supposing that $\{e_1, e_2, \dots, e_n\}$ is the edge set of the graph, let us perturb the weights of the edges as follows:

$$w'_{e_i} := \begin{cases} w_{e_i} - \frac{i}{\alpha}, & \text{if } e_i \in E \\ w_{e_i} + \frac{i}{\alpha}, & \text{if } e_i \notin E \end{cases}$$

where we should imagine α to be a sufficiently large constant which guarantees that the following is true

$$w_{e_i} > w_{e_j} \implies w'_{e_i} > w'_{e_j}. \quad (1)$$

After the perturbation, all edge weights are distinct. Hence, there exists a unique Minimum Spanning Tree of the graph, call it $T' = (V, E')$. We will argue that $T' \equiv T$. Indeed, the cost of T' under the weights w' is

$$\begin{aligned} \text{cost}_{w'}(T') &= \sum_{e_i \in E'} w'_{e_i} = \sum_{e_i \in E' \cap E} w'_{e_i} + \sum_{e_i \in E' - E} w'_{e_i} \\ &= \sum_{e_i \in E' \cap E} \left(w_{e_i} - \frac{i}{\alpha} \right) + \sum_{e_i \in E' - E} \left(w_{e_i} + \frac{i}{\alpha} \right) \\ &= \text{cost}_w(T') - \sum_{e_i \in E' \cap E} \frac{i}{\alpha} + \sum_{e_i \in E' - E} \frac{i}{\alpha} \\ &\geq \text{cost}_w(T) - \sum_{e_i \in E' \cap E} \frac{i}{\alpha} + \sum_{e_i \in E' - E} \frac{i}{\alpha} \quad (\text{since under weights } w \text{ } T \text{ is a MST of graph } G) \\ &\geq \text{cost}_w(T) - \sum_{e_i \in E' \cap E} \frac{i}{\alpha} - \sum_{e_i \in E - E'} \frac{i}{\alpha} \quad (\text{replacing a positive with a negative quantity}) \\ &\geq \sum_{e_i \in E} w_{e_i} - \sum_{e_i \in E} \frac{i}{\alpha} = \sum_{e_i \in E} w'_{e_i} = \text{cost}_{w'}(T). \end{aligned}$$

Hence, T has smaller than or equal cost to T' under the perturbed weights and, since the MST is unique under perturbed weights, it follows that $T' \equiv T$. So under the ordering \mathcal{O} defined by the perturbed weights Kruskal's algorithm returns the desired tree T . We only need to show that \mathcal{O} is a valid ordering under the original weights. But this follows trivially from property (1).

Computing Coulomb Forces - Problem 4 Chapter 5 of Kleinberg and Tardos

Let us consider the following $(2n - 1)$ -length vectors

$$Q = [q_1, q_2, \dots, q_{n-1}, q_n, 0, 0, \dots, 0]$$

$$D = \left[\frac{-1}{(n-1)^2}, \frac{-1}{(n-2)^2}, \dots, \frac{-1}{1^2}, 0, \frac{+1}{1^2}, \frac{+1}{2^2}, \dots, \frac{+1}{(n-1)^2} \right],$$

where by convention their elements are indexed from 0 through $2n - 2$. Let $H := D * Q$ be their convolution; we are interested in the elements H_{n-1} through H_{2n-2} of the convolution. By definition, for all $k \in \{n - 1, \dots, 2n - 2\}$ the k -th element of the convolution is

$$\begin{aligned} H_k &= \sum_{i=0}^k Q_i D_{k-i} = \sum_{i=0}^{n-1} Q_i D_{k-i} && \text{(since } Q_i = 0 \text{ for } i > n - 1) \\ &= \sum_{j=1}^n Q_{j-1} D_{k-j+1} && \text{(change of variables)} \\ &= \sum_{j=1}^n q_j D_{k-j+1} && \text{(since } Q_i = q_{i+1} \text{ for } i \leq n - 1) \\ &= \sum_{j=1}^n q_j D_{(n-1)+\ell+1-j} && \text{(where } \ell = k - (n - 1) \in \{0, \dots, n - 1\}) \\ &= \sum_{j=1}^{\ell} q_j D_{(n-1)+\ell+1-j} + q_{\ell+1} D_{(n-1)+\ell+1-(\ell+1)} + \sum_{j=\ell+2}^n q_j D_{(n-1)+\ell+1-j} \\ &= \sum_{j=1}^{\ell} q_j \frac{1}{((\ell+1) - j)^2} + 0 + \sum_{j=\ell+2}^n q_j \frac{-1}{(j - (\ell+1))^2} \end{aligned}$$

Hence, for all $i \in \{1, \dots, n\}$,

$$H_{(n-2)+i} = \sum_{j=1}^{i-1} q_j \frac{1}{(i-j)^2} - \sum_{j=i+1}^n q_j \frac{-1}{(j-i)^2}.$$

Observe, however, that the force exercised at charge q_i is precisely $q_i \cdot H_{(n-2)+i}$. Hence, if we have the convolution vector H we can compute all forces in $O(n)$ time. Recall that using the FFT we can compute the convolution of two vectors of size $O(n)$ in time $O(n \log n)$. Hence, the whole computation takes time $O(n \log n)$.