HW 4 (due Wednesday, at noon, September 26, 2018)

CS 473: Algorithms, Fall 2018 Version: **1.00**

1 (100 PTS.) Sorting network by merging.

1.A. (10 PTS.) Prove that an n-input sorting network must contain at least one comparator between the ith and $(i+1)$th lines for all $i = 1, 2, ..., n-1$.

Solution:

Just consider the inputs $0^i 1^{n-i}$ and $0^{i-1}101^{n-i}$. Clearly, the second input would be handled correctly only if there is a comparator between the ith wire and $(i+1)$th wire.

1.B. (20 PTS.) Suppose that we have $2n$ elements $< a_1, a_2, ..., a_{2n} >$ and wish to partition them into the n smallest and the n largest. Prove that we can do this in constant additional depth after separately sorting $< a_1, a_2, ..., a_n >$ and $< a_{n+1}, a_{n+2}, ..., a_{2n} >$.
(Hint: Go over the class notes for sorting networks – there is a three line argument here.).

Solution:

This is what the flip-cleaner described in class do. If this was false, then the MERGER – the sorting network for merging two sorted sequences would have failed, since past the flip-cleaner there is no comparator between the top and bottom $n/2$ wires.

1.C. (30 PTS.) Consider a merging network – such a network takes two sorted sequences and merge them into a sorted input. Specifically, in this case, the inputs are $a_1, a_2, \ldots, a_n$, for n an exact power of 2, in which the two monotonic sequences to be merged are $\langle a_1, a_3, \ldots, a_{n-1}\rangle$ and $\langle a_2, a_4, \ldots, a_n\rangle$ (namely, the input is a sequence of n numbers, where the odd numbers or sorted, and the even numbers are sorted). Prove that the number of comparators in this kind of merging network is $\Omega(n \log n)$. Why is this an interesting lower bound? (Hint: Partition the comparators into three sets: One involving inputs only the first half, inputs involving only the bottom half, and comparators involving both halves – then argue one of these sets has to have $\Omega(n)$ comparators, write down a recurrence, and solve it.)

Solution:

To show that we have $\Omega(n \lg n)$ comparators, we can setup a recurrence given the following observation. With such a Odd-even sequence merger, imagine drawing a line horizontally dividing the merger in half. Let your odd sequence be all 1's and your even sequence be all 0's. Since we know that the merger will sort and merge the final output, we know there are at least $n/4$ comparators that crosses this center divided line.(There are n/4 1's in the top half that needs to be trickled down to the second half).
Furthermore, if consider only the top $n/2$ wires, and their input, this is clearly a merging network of size $n/2$ (indeed, imagine giving all the other wires value of $+\infty$). Similarly, this claim holds for the wires/gates involved only the bottom $n/2$ wires. Thus, the number of gates $P(n)$ must follow the recurrence:

$$P(n) \geq 2P(n/2) + n/4 = \Omega(n \lg n)$$

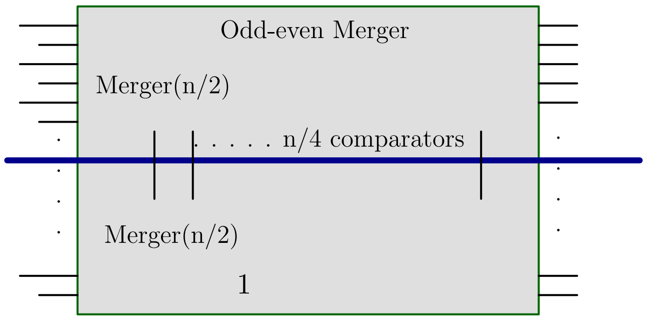

Figure 1: Visualization for proving a lower bound in 2-a

This lower bound is interesting, because it beats the trivial lower bound. Indeed, there are $M = \binom{n}{n/2}$ different outputs, so we need at least $\lceil \log_2 M \rceil$ gates to generate that many different output. However,

$$\log M = \Omega\left(\log \binom{n}{n/2}\right) == \Omega\left(\log \frac{2^n}{\sqrt{n}}\right) = \Omega(n),$$

which is considerably weaker than what we proved in this exercise.

1.D. (40 pts.) Prove that any merging network, regardless of the order of inputs, requires $\Omega(n \log n)$ comparators. (Hint: Use part (C).)

Solution:

Exactly the same argumentation as before works. Indeed, let A be the set of wires that involved the first $n/4$ wires in the first sorted sequence, and the $n/4$ first wires in the second sorted sequence. Let B be all the other wires.

Arguing as above, all the gates and wires on A, form a merging network of size $n/2$. This is less trivial than before. Imagine that we put values $+\infty$ on all the wires of B. Now, consider how the network for this input (where the values on A are two sorted sequences). Let G_A be all the gates in the circuit that both their inputs are both finite. Similarly, let G_B the gates, when we flip the inputs, and all the values entered on A are infinite. Let G_C be all the other gates in the circuit. Arguing as above, $|G_C| \geq n/4$, and both G_A and G_B implement merging networks of size $n/2$. Thus, we get the same recurrence:

$$T(n) \geq |G_C| + |G_A| + |G_B| \geq n/4 + 2T(n/2) = \Omega(n \log n).$$

2 (100 pts.) Reorderings.

A *permutation network* on n inputs and n outputs has switches that allow it to connect its inputs to its outputs according to any $n!$ possible permutations. Figure 2 shows the 2-input, 2-output permutation network P_2, which consists of a single switch that can be set either to feed its inputs straight through to its outputs or to cross them.

2.A. (5 pts.) Argue that if we replace each comparator in a sorting network with the switch of Figure 2, the resulting network is a permutation network. That is, for any permutation π, there is a way to set the switches in the network so that input i is connected to output $\pi(i)$.

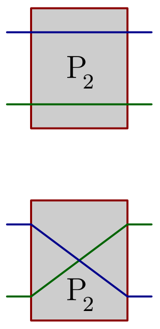

Figure 2: Permutation Switch

Solution:

Let in_1, in_2 be the input to a comparator. If $in_1 < in_2$, we replace the comparator with a switch set to passing the inputs straight through. If $in_2 < in_1$, we replace the comparator with a switch set to cross the inputs.

Observe that since in a permutation network, each input is directly connected to each output. Interchanging the output and input will generate the same network.

Since the sorting network will sort any sequence, we can sort any permutation π. Therefore, we can generate any permutation network by using the above defined operators when the permutation is feed through the sorting network. Essentially these operators configure the network to map $\langle 1, 2, \ldots, n \rangle$ to $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$.

2.B. (2 PTS.) Figure 3 shows the recursive construction of an 8-input, 8-output permutation network P_8 that uses two copies of P_4 and 8 switches. The Switches have been set to realize the permutation $\pi = \langle \pi(1), \pi(2), \ldots, \pi(8) \rangle = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$, which requires (recursively) that the top P_4 realize $\langle 4, 2, 3, 1 \rangle$ and the bottom P_4 realize $\langle 2, 3, 1, 4 \rangle$.

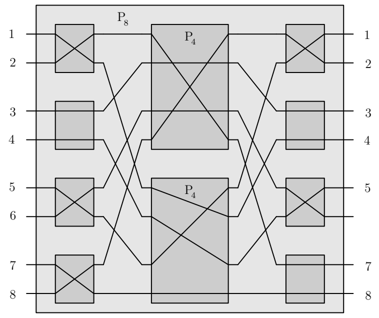

Figure 3: Permutation Network of Size 8

Show how to realize the permutation $\langle 5, 3, 4, 6, 1, 8, 2, 7 \rangle$ on P_8 by drawing the switch settings and the permutations performed by the two P_4's.

Solution:

The permutation realized by top P_4 is $\langle 3, 2, 1, 4 \rangle$. and the permutation realized by the bottom P_4 is $\langle 2, 3, 4, 1 \rangle$.

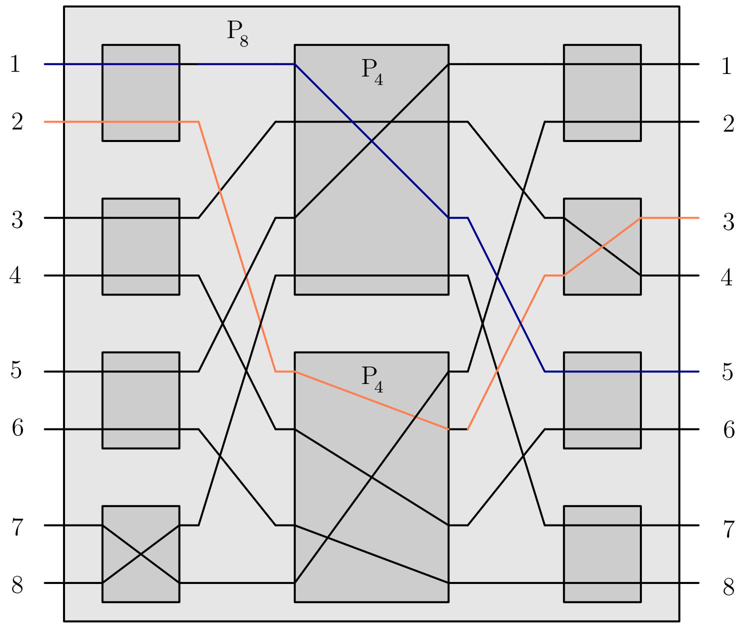

Figure 4: Solution to 3-b

2.C. (5 PTS.) Let n be an exact power of 2. Define P_n recursively in terms of two $P_{n/2}$'s in a manner similar to the way we defined P_8.

Describe an algorithm (ordinary random-access machine) that runs in $O(n)$-time that sets the n switches connected to the inputs and outputs of P_n and specifies the permutations that must be realized by each $P_{n/2}$ in order to accomplish any given n-element permutation. Prove that your algorithm is correct.

Solution:

First let us define the notion of a conflict. A conflict is given a set of π values, if there exist a tuple $(\pi(i), \pi(j)) = (2k, 2k-1)$ or $(2k-1, 2k)$ for some positive integer k.

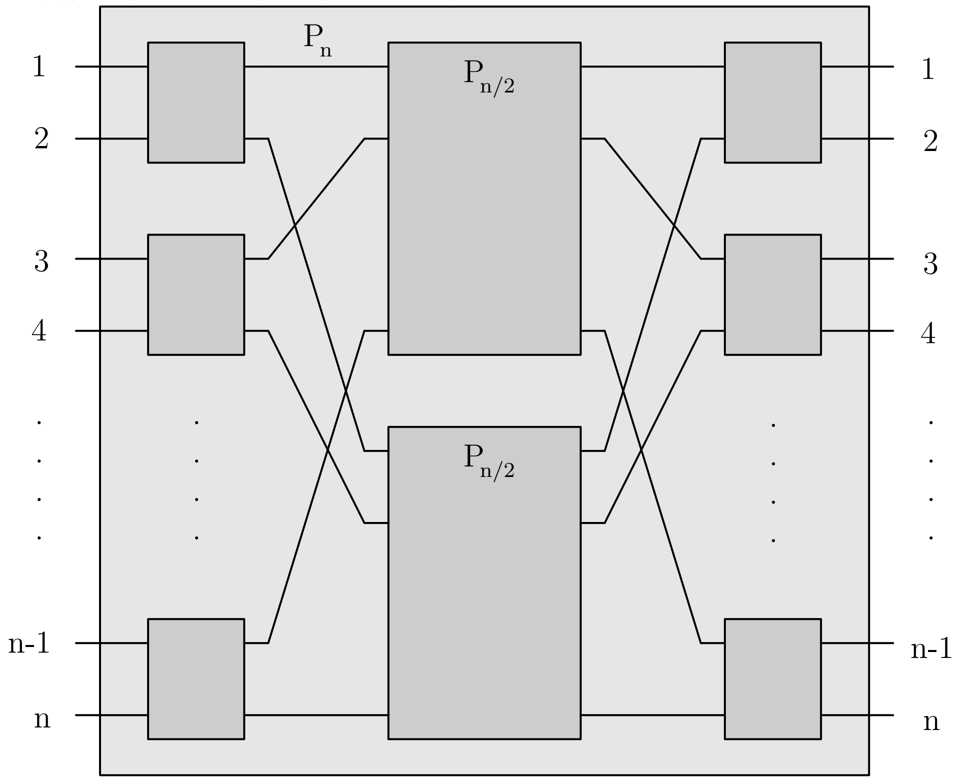

Figure 5: Solution to 3-c

Observe that the output of each pair of $n/2$ input switches must be separated into the 'top' $P_{n/2}$ and the 'bottom' $P_{n/2}$. Also notice that the each pair of inputs to the $n/2$ output switches cannot come from the same $P_{n/2}$ either.

The algorithm to set the input switches is just now partitioning the π values appropriately into the two 'top' and 'bottom' $P_{n/2}$ without introducing any conflicts in either 'top' or 'bottom' $P_{n/2}$. Since the $P_{n/2}$ simply sorts the input π values, and since we also know the ordering of the fixed wire in between the $P_{n/2}$'s and the output switches, we can now set the output switches accordingly to realize the sorted $\pi(i)$'s configuration. To show that this algorithm is correct, is equivalent to showing that we can partition the $\pi(i)$'s successfully without conflict. Observe that if there exist a conflict in the 'top' $P_{n/2}$ and a conflict in the 'bottom' $P_{n/2}$, we can eliminate these conflict by swapping conflicting $\pi(i)$'s until no more conflicts are left. Suppose there is just one conflict in the 'top' $P_{n/2}$, this means that there exist two values that came from, or lead to the same switch. With n inputs(outputs), there are exactly $n/2$ inputs(outputs) for each $P_{n/2}$. Let each pair of inputs for each switch be $\langle a_1, b_1\rangle \ldots \langle a_{n/2}, b_{n/2}\rangle$ where the subscript represent which switch the inputs are from. Then if a_k and b_k are in the same $P_{n/2}$ that means there must be some input from another switch a_j and b_j that are in the other $P_{n/2}$. Since we have two conflicts, this can be resolve. Therefore we can partition these values successfully without conflicts.

Partitioning the inputs take $O(n)$ and setting the output switches also takes $O(n)$ time, therefore, total complexity is $O(n)$.

Alternative Approach: Imagine each input to the network is represented by a vertex. Since we know that each of the two inputs values from the same switch cannot be in the same grouping($P_{n/2}$), we can connect the two input values for every switch by an edge in a (separate graph) G. For example, if a_1 and a_2 are inputs to the same switch, we will connect 1 and 2 with an edge in G. Similarly, since the inputs to the same output switch (the switches on the last column of the circuit) must come from a different $P_{n/2}$, we can draw an edge between these two values. For example, if a_7 and a_15 are inputs to the same output switch, we draw an edge between 7 and 15 in G. Notice that each vertex in G has degree two (since it participates in exactly two switches), we have a graph G which is just the union of cycles. Furthermore, all those cycles are of event length (as can be easily verified). And trivially, we know that each of these cycles are two colorable. Now we can arbitrarily two color the graph, each color representing one of the $P_{n/2}$. This enable us to set all the input switches. This guarantees us that there will be no conflicts. Now since we know the ordering of the output, it is trivial to set the output switches. Since coloring the graph take linear time and setting the outputs switches also take linear time, this algorithm runs in $O(n)$.

Figure 6 gives a simple example that realizes the permutation $\langle 8, 3, 2, 5, 1, 7, 4, 6\rangle$

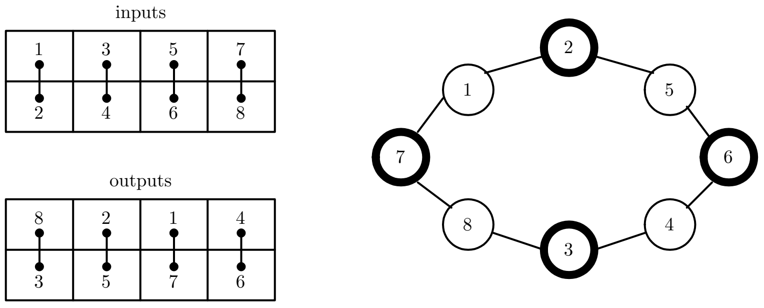

Figure 6: Simple example for the Graphical Approach: Each column in the table represents a switch, therefore no two values in the same column can be in the same set. Each of the two values in each column are connected by an edge, forming the graph. The graph is two colored allowing the inputs to be partitioned.

2.D. (3 PTS.) What are the depth size of P_n? How long does it take on an ordinary random-access machine to compute all switch settings, including those within the $P_{n/2}$'s?

Solution:

The recurrence for the depth is $D(P_n) = D(P_{n/2}) + 2 = \Theta(\lg n)$ This is because the outer switches causes an extra 2 level of depths.

The recurrence for setting the total number of switches is $S(P_n) = 2S(P_{n/2}) + O(n)$, which is just $\Theta(n \lg n)$. Setting the outer switches takes linear time from part c, and thus the running time is $T(n) = O(n) + 2T(n/2) = O(n \log n)$.

2.E. (5 PTS.) Argue that for $n > 2$, any permutation network (not just P_n) must realize some permutation by two distinct combinations of switch settings.

Solution:

Observe that in any permutation network with n inputs, there must be at least $M = \lceil \lg(n!) \rceil$ switches (since, there are $n!$ possible outputs to the circuit). Therefore, there are 2^M ways to set switches. Since there are n inputs, there are only $n!$ different permutations.

Thus, there are two different settings that lead to the same permutation if 2^M is strictly larger than $n!$. But this obviously true, since for $n > 2$, $n!$ is not a power of two. As such, it must be that $2^M > n!$.

3 (100 PTS.) Two is enough

Consider a uniform rooted tree of height h (every leaf is at distance h from the root). The root, as well as any internal node, has, say, 4 children. Each leaf has a boolean value associated with it. Each internal node returns true, if two or more of its children are one. The evaluation problem consists of determining the value of the root; at each step, an algorithm can choose one leaf whose value it wishes to read.

3.A. (35 PTS.) [This part is a challenging question – lets see what you can do.] Show that for any deterministic algorithm, there is an instance (a set of boolean values for the leaves) that forces it to read all $n = 4^h$ leaves. (Hint: Consider an adversary argument, where you provide the algorithm with the minimal amount of information as it requests bits from you. In particular, one can devise an adversary algorithm that forces you to read all input bits. And naturally, first solve the cases $h = 1$, $h = 2$, etc. Partial credit would be given for solving the cases $h = 1, 2, 3$.)

Solution:

The construction is by induction on height. Consider a tree of height 1. For the first two leafs read, the algorithm return their values as 0, the third leaf read it returns as 1, and the last one, it returns according to whatever value we want the root to have. We will refer to the last leaf read, as the ***critical*** read, since it determines the value of the tree.

So, assume we had an implementation of such an adversary A_h, which forces the user to read all leafs of a tree before the user can resolve the values of the tree – furthermore, the last value read is the value of tree. Given a tree of height $h + 1$, the adversary A_{h+1} would run four copies of A_h for each subtree. Given a request for a leaf value, it would send the request to the relevant adversary. Given a critical read (i.e., the last leaf read in one of the four subtrees) – there are four such critical reads – we tread them as the case for $h = 1$, and use the strategy described above. Clearly, the value the tree can be determined only after four critical reds were done, but that implies that leafs had been read.

3.B. Consider a tree T with $h = 1$. Consider the algorithm that permute the children of the root of T, and evaluate them by this order. The algorithm stops as soon as one can compute the value of the root (for example, the algorithm encountered two ones). Let X be the number of leafs this algorithm evaluates. Prove that $\mathbb{E}[X] < 4$. What exactly is your upper bound for $\mathbb{E}[X]$?

(Hint: There are really only 5 different inputs: Calculate $\mathbb{E}[X]$ for each one of these scenarios, and return the weakest upper bound you found.)

Solution:

The two "hard" inputs are $1, 1, 0, 0$ and $1, 0, 0, 0$. For the case that $I = 1, 1, 0, 0$, there are $\binom{4}{2}$ different orderings of the input, which all have the same probability to be read in this order: $(1, 1, 0, 0), (1, 0, 1, 0), (1, 0, 0, 1), (0, 1, 1, 0), (0, 1, 0, 1), (0, 0, 1, 1)$. A such

$$\begin{aligned}\mathbb{E}[X \mid I = 1,1,0,0] = {} & 2 \cdot \mathbb{P}[\text{read two 1s in two bits read}] + 3 \cdot \mathbb{P}[\text{read second 1 in third read}] \\ & + 4 \cdot \mathbb{P}[\text{read second 1 in last read}] \\ & 2 \cdot \frac{1}{6} + 3\frac{2}{6} + 4\frac{3}{6} = \frac{20}{6} = 3\frac{1}{3}.\end{aligned}$$

Similarly, we have

$$\mathbb{E}[X \mid I = 1,0,0,0] = 3 \cdot \frac{1}{4} + 4 \cdot \frac{3}{4} = 3\frac{3}{4}.$$

As such, $\mathbb{E}[X] \leq 3.75$ in the worst case.

3.C. Consider the recursive randomized algorithm that evaluates (recursively) the subtrees of a node in a random order, and stops as soon as the value of the node is resolved. (Unlike part (A), the specific input is fixed in advance – there is no adversary here.) Show the expected number of leaves read by the algorithm on any instance is at most n^c, where c is a constant strictly smaller than 1,

Solution:

Let $L(h)$ be the expected number of leafs the above algorithm reads on a tree of height h. By part (B), we have that $\mathbb{E}[L(1)] \leq 3.75$. More generally, if we have a tree of height h, the above algorithm would require, in expectation, at most 3.75 evaluations of trees of height $h-1$. More precisely, let X be the random variable which is the number of children of the root that the algorithm evaluated. We have that the expected number of leafs read by the algorithm is

$$\mathbb{E}[X \cdot L(h-1)] = \mathbb{E}[X]\,\mathbb{E}[L(h-1)] \leq 3.75 \cdot \mathbb{E}[L(h-1)] \leq (3.75)^h,$$

where we used that X and the number of evaluations needed in a subtree are independent variables. Now, $n = 4^\ell$, for some ℓ. As such, the algorithm in expectation evaluates at most

$$3.75^\ell \leq 4^{\ell \log_4 3.75} = n^{\log_4 3.75} \approx n^{0.953\cdots},$$

as claimed.