18.435/6.848 Theory of Parallel and VLSI Computation **Fall 1997**

Solutions to Assignment #5

TA: Daniel Lewin *Prof. Leighton*

Problem 1

First let's show that $\omega = 2$ is a principal Nth root of unity in $\mathbb{Z}_{2^N-1}$. We have $\omega^N = 2^N = (2^N - 1) + 1 = 1 \bmod (2^N - 1)$. We cannot have $\omega^k = 1 \bmod (2^N - 1)$ for any $k \in [1, N-1]$, since then $\omega^k \in [2, 2^{N-1}]$, and so $\omega^k \bmod (2^N - 1) \in [2, 2^{N-1}]$.

Take any even N. Let $A = FF^{-1}$. Let's compute entry $a_{\frac{N}{2},0}$ of A.

$$a_{\frac{N}{2},0} = \sum_{k=0}^{N-1} \omega^{\frac{N}{2}k} \cdot \omega^{-k\cdot 0}/N = \frac{1}{N}\sum_{k=0}^{N-1} (2^{\frac{N}{2}})^k = \frac{1}{N}\left(\sum_{k'=0}^{\frac{N}{2}-1} (2^{\frac{N}{2}})^{2k'} + \sum_{k'=0}^{\frac{N}{2}-1} (2^{\frac{N}{2}})^{2k'+1}\right)$$

$$= \frac{1}{N}\left(\sum_{k'=0}^{\frac{N}{2}-1} (2^N)^{k'} + 2^{\frac{N}{2}} \cdot \sum_{k'=0}^{\frac{N}{2}-1} (2^N)^{k'}\right) = \frac{1}{N} \cdot \frac{(1+2^{\frac{N}{2}})N}{2} = (2^{\frac{N}{2}}+1)\cdot 2^{-1} \pmod{2^N - 1}.$$

Suppose F^{-1} is the inverse of F. Then $A = I$ and $a_{\frac{N}{2},0} = 0 \pmod{2^N - 1}$ (since $\frac{N}{2} \neq 0$). Then $2a_{\frac{N}{2},0} = 2^{\frac{N}{2}} + 1 = 0 \pmod{2^N - 1}$. This, however, is impossible, since $0 < 2^{\frac{N}{2}} + 1 < 2^N - 1$. Contradiction.

Problem 2

We prove this by induction on r. For $r = 0$ there is only one input and one output, so the result is obvious. Assume the result is true for an $(r-1)$-dimensional Benesnetwork.

The middle $2r - 1$ levels of an r-dimensional Benesnetwork comprise two $(r-1)$-dimensional Benesnetworks. Hence, it will be sufficient to decide whether each path is to be routed through the upper sub-Benesnetwork or the lower sub-Benesnetwork. The only constraints we have are that paths from inputs i and $i + 2^{r-1}$ must use different sub-networks for $1 \leq i \leq 2^{r-1}$ and that paths to outputs i and $i + 2^{r-1}$ must use different subnetworks.

We can assign the each path to the upper or lower subnetwork as follows. We start by routing the first path through the upper subnetwork. We next satisfy the constraint generated in the output by routing the corresponding path through the lower subnetwork. We next satisfy the constraint generated at the input by routing the appropriate path through the upper subnetwork. We keep on going back and forth through the network, satisfying constraints at the inputs by routing through the upper subnetwork and satisfying constraints at the outputs by routing through the lower subnetwork. Eventually we will close the loop by routing a path that starts at some node i through the lower subnetwork (in response to an output constraint) such that the first path that was routed starts at $i \pm 2^{r-1}$. Since the first path that was routed used the upper subnetwork, the original input constraint is now also satisfied. If any additional paths need to be routed, we continue as before, starting over again with an arbitrary unrouted path. In this way, all paths can be assigned to the upper or lower subnetworks without conflict (i.e., we can set the switches at the first and last levels of the Benesnetwork so that both ends of every path are connected to the same subnetwork). The remainder of the path routing and switch setting is handled by induction in the subnetwork. Hence, we have established the inductive hypothesis, thereby proving the theorem.

Problem 3

Node disjoint paths for the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 8 & 1 & 3 & 4 & 7 & 6 & 2 \end{pmatrix}$ are shown in fig. PS5.1.

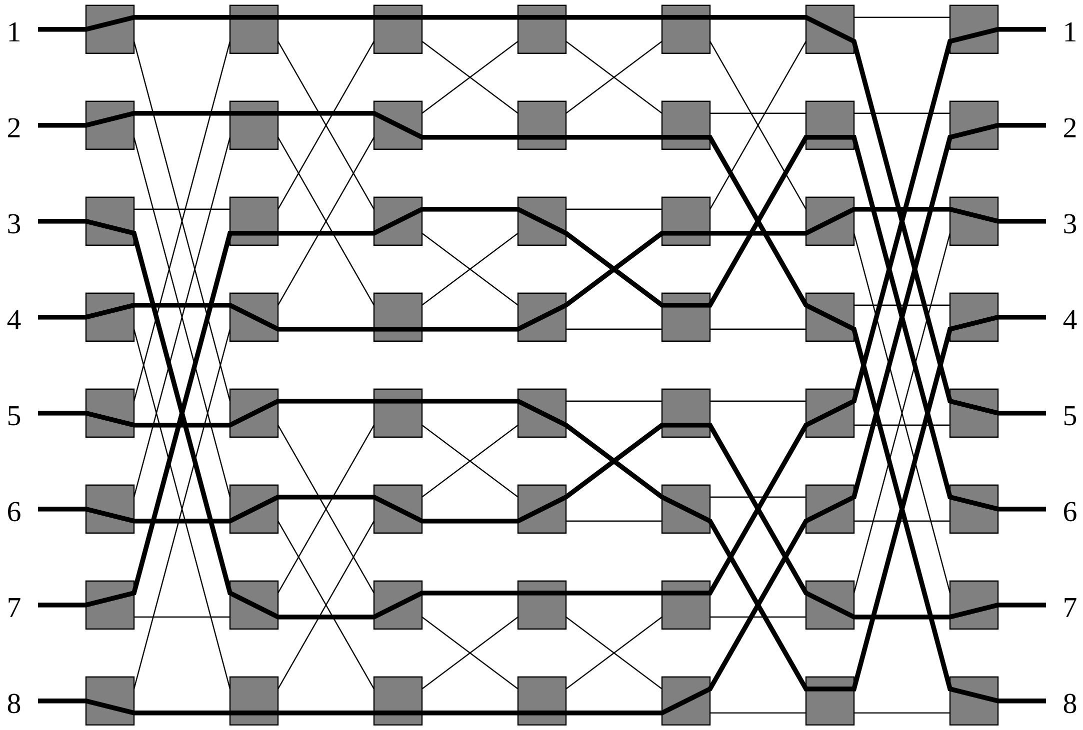

Figure PS5.1: Node disjoint paths in a 3-dimensional Benesnetwork

Problem 4

By the 0-1 sorting lemma, we only need to consider lists of 0's and 1's. It is easy to see that a list of 0's and 1's is bitonic iff it is of the form $1^i0^j1^k$, $i \geq 0$, $j \geq 0$, $k \geq 0$, or if it is of the form $0^i1^j0^k$, $i \geq 0$, $j \geq 0$, $k \geq 0$. Assume the list has length 2^r (since an r-dimensional butterfly comparator circuit has 2^r inputs). Let's show by induction on r that the butterfly comparator circuit will sort a bitonic list of 0's and 1's.

The case $r = 0$ is obvious, since nothing needs to be sorted when there is only one input.

Suppose the claim is correct for $r - 1$. Let's show it for r. For a list L of even length, denote by $\imath first(L)$ the first half of L and by $\imath second(L)$ the second half of L. Denote by $\imath reverse(L)$ the result of reversing L. For lists A and B (not necessarily sorted), denote by $\imath oe - merge(A, B)$ the result of running odd-even merge on A and B.

We will need the following

Lemma 0.1 *For any* $r > 0$, $i \geq 0$, *and* $j \geq 0$, $\imath oe - merge(1^i0^{2^{r-1}-i}, 1^j0^{2^{r-1}-j}) = 0^{2^r-i-j}1^{i+j}$.

Proof: We will show this by induction on r. The case $r = 1$ is obvious.

Suppose the claim is true for $r - 1$. Let $A = 1^i0^{2^{r-1}-i}$ and $B = 1^j0^{2^{r-1}-j}$. $\imath even(A)$ has $\lceil \frac{i}{2} \rceil$ 1's, $\imath odd(A)$ has $\lfloor \frac{i}{2} \rfloor$ 1's, $\imath even(B)$ has $\lceil \frac{j}{2} \rceil$ 1's, $\imath odd(B)$ has $\lfloor \frac{j}{2} \rfloor$ 1's, and all four of these are in reverse sorted order. By the inductive assumption, $C = \imath oe - merge(\imath even(A), \imath odd(B))$ is in sorted order and has $\lceil \frac{i}{2} \rceil + \lfloor \frac{j}{2} \rfloor$ 1's. Also, $D = \imath oe - merge(\imath odd(A), \imath even(B))$ is in sorted order and has $\lfloor \frac{i}{2} \rfloor + \lceil \frac{j}{2} \rceil$ 1's. The number of 1's in C and D differs by at most 1. Now, just like in the proof of correctness for odd-even merge sort, we obtain that $\imath oe - merge(A, B)$ is in sorted order. ☐

Let L be a bitonic list of 0's and 1's of length 2^r. Note that nodes of levels 2 through r of an r-dimensional butterfly comparator circuit form two $(r - 1)$-dimensional sub-butterfly comparator circuits. We need to consider the following cases.

1. $L = 0^i1^j0^{2^r-i-j}$, $i \geq 2^{r-1}$

   In this case $\imath first(L) = 0^{2^{r-1}}$. Since for each $l \in [0, 2^{r-1}]$ we have $L_{l+2^{r-1}} \geq 0 = L_l$, we see that the upper sub-comparator circuit will get all 0 inputs, while the lower sub-comparator circuit will get inputs $0^{i-2^{r-1}}1^j0^{2^r-i-j}$. The upper sub-comparator will obviously output all 0's. The lower sub-comparator, by the inductive assumption, will sort its inputs and output $0^{2^{r-1}-j}1^j$. The output of the whole circuit will be $0^{2^r-j}1^j$, which is in sorted order.

2. $L = 0^i1^j0^{2^r-i-j}$, $i + j \leq 2^{r-1}$

   This case is handled analogously to the previous one.

3. $L = 0^i1^j0^{2^r-i-j}$, $i < 2^{r-1}$, $i + j > 2^{r-1}$

   In this case $\imath first(L) = 0^i1^{2^{r-1}-i}$ and $\imath second(L) = 1^{i+j-2^{r-1}}0^{2^r-i-j}$. We know that the butterfly comparator circuit implements odd-even merge of the first half of the inputs with the reverse of the second half, i.e., the result of the circuit is equal to $\imath oe - \mathsf{merge}(\imath\mathsf{first}(L), \imath\mathsf{reverse}(\imath\mathsf{second}(L)))$. In this case, it will be $\imath oe - \mathsf{merge}(0^i 1^{2^{r-1}-i}, 0^{2^r-i-j} 1^{i+j-2}$ Note that both inputs to ıoe-merge are in sorted order. Since ıoe-merge will merge two sorted lists into a sorted list, we see that the result will be in sorted order.

4. $L = 1^i 0^j 1^{2^r-i-j}$, $i \geq 2^{r-1}$ or $i + j \leq 2^{r-1}$

   This case is handled similarly to the first two, except here the lower sub-comparator will get all 1 inputs and the upper one will get a bitonic list.

5. $L = 1^i 0^j 1^{2^r-i-j}$, $i < 2^{r-1}$, $i + j > 2^{r-1}$

   In this case the result of the circuit will be $\imath oe - \mathsf{merge}(1^i 0^{2^{r-1}-i}, 1^{2^r-i-j} 0^{i+j-2^{r-1}})$. It follows from the lemma above that the output will be in sorted order.

Problem 5

The paths for the packing problem are shown in fig. PS5.2.

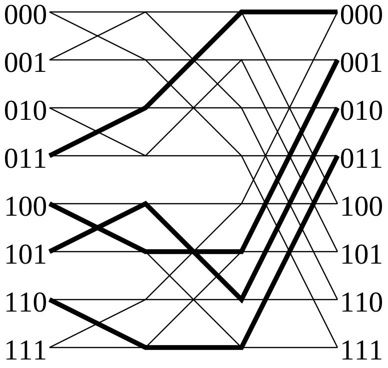

Figure PS5.2: Greedy paths for a packing problem

If we traverse the levels in the opposite order, the greedy algorithm fails for packing with inputs 100 and 110: the paths are not node disjoint (see fig. PS5.3).

Problem 6

Here is the algorithm:

1. Offline: Compute the permutation needed to sort the packets based on their destination address. Compute node-disjoint paths in the Benesnetwork that correspond to this permutation.
2. Simulate the Benesnetwork with two passes through the butterfly: from level 0 to level r and back to level 0 (in $2 \log N$ steps). Now the packets are sorted.

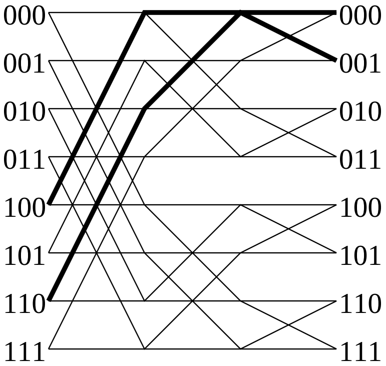

Figure PS5.3: Greedy paths for a packing problem with reverse traversal

3. Use parallel prefix to set up barriers between groups of packets that are addressed to different destinations ($O(\log N)$ steps).
4. Use segmented parallel prefix to combine the packets in each group ($O(\log N)$ steps).
5. Let every node which does not have a barrier right before it drop its packet (1 step).
6. We now have a monotone routing problem, which, as we know, can be solved in $O(\log N)$ steps.

The algorithm takes $O(\log N)$ steps.

Now let's show how to handle one-to-many packets. Since $m_1 + m_2 + \cdots + m_N \leq cN$ for some constant c, we can assume wlog that $m_1 + m_2 + \cdots + m_N \leq N$ (if $c > 1$, we can arbitrarily select some copies at each node so that the total is N and route them, then select N of the rest, etc., for a total of at most $\lceil c \rceil$ phases, each taking $O(\log N)$ time). We can use the following algorithm:

1. Pack the inputs (in $O(\log N)$ steps).
2. Use parallel prefix to compute for each node the sum of the number of copies for the nodes above (in $O(\log N)$ steps).
3. Do spreading, using the sums computed above as the destinations ($O(\log N)$ steps).
4. Make the copies using segmented parallel prefix, with a segment barrier before each node containing a packet ($O(\log N)$ steps).
5. Use segmented parallel prefix to compute each node's position in its segment ($O(\log N)$ steps).

6. Now each node knows the destination for the packet it is holding. We have a many-to-one routing problem, which can be solved in $O(\log N)$ steps by the algorithm above.

This algorithm runs in $O(\log N)$ steps.

We can route N packets in $O(\log N)$ steps on an N-node shuffle-exchange graph by simulating the above butterfly algorithm (we will get a slowdown of 2). The algorithm is clearly normal.

Problem 7 We can sort N packets of $O(1)$-bit size based on 1-bit keys (preserving relative order for packets with equal keys) as follows:

1. Consider the packets with 0 keys as inputs to a packing problem. We can solve this packing problem in $O(\log N)$ bit steps (note that a packet does not need to fully know its destination at each step, only the next bit of it, so we can use 1-bit adders in the parallel prefix computation).
2. Route the packets with 1 keys to consecutive positions up to N, preserving their relative order. It can be shown just like for packing that the greedy algorithm works here in $O(\log N)$ time. Once again, we can use parallel prefix with 1-bit adders to tell packets what edge to take at each step. The total number of bit steps will be $O(\log N)$.

Note that these problems can be routed in parallel (assuming $O(1)$-bit-size buffers and $O(1)$-bit-capacity channels), and they can both be pipelined. This way we can sort N k-bit numbers on any bit position in $O(\log N + k)$ bit steps. We can then repeat this process k times to do radix sort in $O(k(\log N + k)) = O(k \log N)$ bit steps.

Problem 7

First we see how to solve this problem on an N-input butterfly. Each input node i starts with k_i packets where $k_i \leq k$, and there are N total packets. Each packet can be assigned a unique destination as follows: Each input node i computes k_i by counting how many packets it has. Next each input node i finds the value $x_i = \sum_{j<i} k_j$; this can be done with a prefix computation in $O(\log N)$ steps. Finally each input node i assigns each of its k_i packets a destination by giving its jth packet the destination $x_i + j$; this takes $O(k)$ steps. Thus in $O(k + \log N)$ steps each packet gets a unique destination. Now we just have k monotone routing problems each of which can be solved in $O(\log N)$ steps, so by pipelining we solve the entire problem in $O(k + \log N)$ steps.

Now we can run this algorithm on an N-node hypercube by having each hypercube node simulate the operation of the corresponding row in the butterfly. At each step of the butterfly algorithm each row contains no more than k packets and our hypercube has queues of size k; furthermore our hypercube can process a packet on each of its edges at every step. Thus the simulation incurs no slowdown, and our hypercube algorithm runs in $O(k + \log N)$ steps as desired.

Problem 8

We want to sort kN items given in blocks of k sorted elements each using a comparison based sorting algorithm for sorting N items in which the compare-exchange operations have been replaced with merge/split operations (in the following discussion we will refer to both types of operations as comparator operations). Since we are dealing with an oblivious comparison-exchange algorithms, we can use the $0 - 1$ lemma, i.e. we can prove the required results using blocks of 0's and 1's.

For clairity we assume that the algorithm is given as a sorting circuit $\mathcal{C}$. In this setting the compare-exchange *nodes* are replaced by merge-split *nodes*.

We prove that the circuit works on kN items by contradiction. In other words, we will suppose that $\mathcal{C}$ fails to sort some input of kN items. In this case, we will construct an instance of the N input problem on which $\mathcal{C}$ will also fail. Now, the "details".

We use the following notation:

S - a solid block, i.e. block where the k items are either all 0 or all 1.

S_1 - a solid block of 1's.

S_0 - a solid block of 0's.

M - a mixed block, i.e. block containing some 0's and some 1's.

M_i - a mixed block with i 0's.

$C[X, Y] = [U, W]$ - the result of performing a merge/split on blocks X and Y is U and W. Note that when we don't care about the order of U,W, we will write $C[X, Y] = \{U, W\} = \{W, U\}$

We make the following fairly trivial observations.

1. $C[S_i, S_j] = [S_{i \wedge j}, S_{i \vee j}]$, i.e. if we input two solid blocks, we output two solid blocks.
2. $C[S_i, M] = \{M, S_i\}$, i.e. if we input a solid and a mixed block, we get the same back.
3. $C[M_i, M_j] = [M_{i+j}, S_1]$ if $i + j < k$; $C[M_i, M_j] = [S_0, M_{i+j-k}]$ if $i + j > k$; $C[M_i, M_j] = [S_0, S_1]$ if $i + j = k$. In other words, if we input two mixed blocks, at least one of the output blocks is solid.

Suppose that given a local computation (a comparator node, two inputs and two outputs) we wanted to change the outputs. We want to figure out how we can propagate these changes to previous levels of computation in the $\mathcal{C}$.

Claim 0.2 *We can take all the mixed blocks at the output of $\mathcal{C}$ and make them arbitrarily into solid blocks propagating any changes back to the input blocks so that the input blocks are all solid.*

Proof:

We do this iteratively. In other words, we first make the output(last level) blocks all solid and propagate these changes to the previous levels of $\mathcal{C}$, then we make all the blocks on the (last-1) level all solid, propagating the changes backward, and so forth. We continue in this fashion until we have made all the input blocks solid. In the following discussion, when we say that we make a bunch of changes to a block, like changing a mixed block to be a solid block, we mean that we change one bit at a time. In other words, we change one bit in the block, propagate this change until it reaches the input level, change the next bit and propagate that change, etc.

Look at any output block that was mixed (M_i) and has now been changed to (wlog) S_1. Let C_o be the comparator whose output was M_i, and let I_{o_1} and I_{o_2} be the two inputs to C_o. Note that by observation 3, the other output of C_o must have been a solid block. From our observations, we know that either (a) both of I_{o_1} and I_{o_2} were mixed, or (b) one of I_{o_1} or I_{o_2} was mixed, say I_{o_1}. In case (b), when we change M_i to be S_1, we simply change I_{o_1} to be S_1 and propagate the change to I_{o_1} to the previous levels. In case (a), for each 0 of M_i that was changed into a 1, we change one of the 0's of either I_{o_1} or I_{o_2} to a 1 and propagate the changes. Unfortunately, it may not be possible to make both I_{o_1} and I_{o_2} into solid blocks by the previous step. For example, suppose that $k = 3$ and the input blocks are 001 and 001. The output then is 000 and 011. But by changing the output 011 to be the solid block 111 we only change one 0 to one 1 and it is obvious that there is no way to propagate this change through C_o so that either of the inputs is solid. Fortunately, after we propagate the change of M_i to S_1 through C_o, we *do* know that either both inputs are now solid (since one solid input and one mixed input cannot yield two solid outputs) or both are now mixed. If we can make both inputs solid (consider the example of inputs 011 and 011), then we are done - i.e. we simply propagate the changes to I_{o_1} and I_{o_2}. If both are mixed, we know that after the changes to M_i have been propagated to I_{o_1} or I_{o_2}, the number of $1'$s in I_{o_1} must equal the number of O's in I_{o_2} and vice versa. Otherwise, how could we have two solids as the output of I_{o_1} or I_{o_2}! Thus, we can make them solid by pairing up the 0's and the 1's in the two inputs, i.e. by switching each 1 of one of them, say I_{o_1} to a 0 and at the same time switching each 0 or I_{o_2} to a 1, and propagating these changes to the previous levels. In doing this, we have not changed the overall ratio of 1's to 0's. Moreover, this pairing does not effect the output blocks of the C_o since we are keeping the total number of 0's at the input constant (and trivially, the total number of 1's at the input also remains constant).

Once we have changed the output blocks to be all solids, we work on the next level using the algorithm just described. One important observation needs to be made – when we are running the above algorithm on some level j of $\mathcal{C}$, the changes that the algorithm makes effect only levels $j - 1$ through 1 (where 1 is the initial input level). In other words, once we have fixed some level j to be all solids and we recurse on the previous levels, we never "mess up" level j, in fact we never even touch it. This observation follows from the pairing argument we made in the previous paragraph.

□

Now, suppose $\mathcal{C}$ failed to sort some kN inputs. That means that we have more than one mixed block in the output. Using the claim, change the first mixed block, say the i-th block in the overall sorted order (assuming that the sorted order is a bunch of 0's followed by 1's) into all 1's and the second mixed block, say the j-th block, into all 0's. Change the other mixed blocks to either of the solids. By our claim, the changes can be propagated so that all the blocks at each level of $\mathcal{C}$ are all solid. Note there there is an exact correspondence between a circuit working on blocks of inputs where at all the stages all the blocks are solid and the same circuit (mod changing the merge/split comparators to the compare-exchange comparators) running on single inputs. In other words, we can change this kN input problem to an N input problem by making each S_0 into a 0 and each S_1 into a 1 and we know that $\mathcal{C}$ behaves the same way on both problems. Thus, $\mathcal{C}$ will fail to sort this input, i.e. output i will be 1 and output j, $i < j$, will be 0. A contradiction.

Problem 10

Edges are directed so that packets traveling in opposite directions do not interfere with each other.

Consider an edge e on the line $l = (v_1, v_2, \ldots, v_{i-1}, *, v_{i+1}, \ldots, v_k)$.

Figure PS5.4: The line $l = (v_1, v_2, \ldots, v_{i-1}, *, v_{i+1}, \ldots, v_k)$

Consider a packet which crosses edge e during routing. Its origin must be of the form: $(*, *, \ldots, *, s, v_{i+1}, \ldots, v_k)$ where $s \leq j$ and its destination must be of the form: $(v_1, v_2, \ldots, v_{i-1}, d, *, *,$ where $d \geq j + 1$.

The number of packets having this type of origin is $pm^{i-1}j$.

The number of packets having

Problem 12

The important idea to solve this problem is that the $O(\log N)$-step zero-one sorting algorithm described in class is stable — that is, the relative orders of the zeros which are packed to the left and the ones which are packed to the right are unchanged. this is precisely what we need to implement radix sort on a butterfly.

The algorithm works in k pipelined phases. In phase i, the numbers are sent, the ith lowest order bit first, through the butterfly, packing the elements with bit i being 0 to the left, and those with bit i being 1 to the right; the other bits follow this path through the network. When the first bits reach their destinations, each number holds back bit i to send it through last in the next phase, so that bit $i+1$ is at the head of the number for the next phase. Since each phase preserves the relative order of the lower-order bits (and therefore the sorted order so far), after this procedure is repeated for each of the k bits the numbers will be sorted. Each phase needs $O(\log N)$ steps before the next phase can be started, and there are a total of k phases. Thus the total running time of the algorithm is $O(k \log N)$.