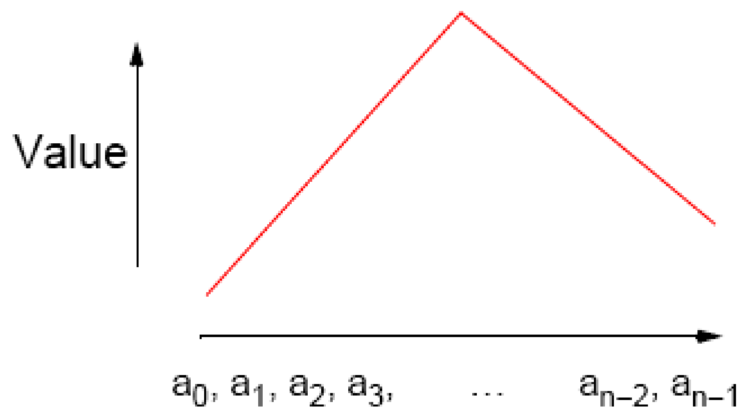
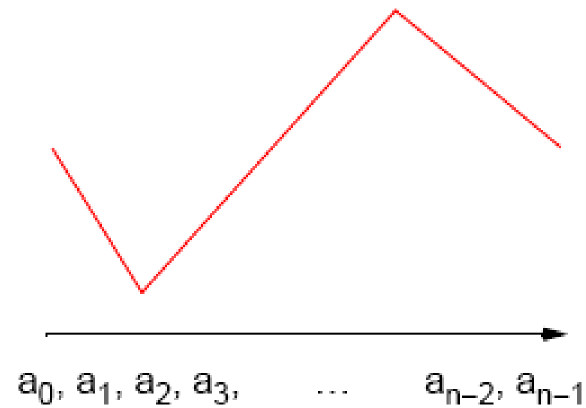
GPU: Parallel bitonic sort

Bitonic Sequence

A *bitonic sequence* is defined as a list with no more than one **LOCAL MAXIMUM** and no more than one **LOCAL MINIMUM**. Endpoints are considered wraparound.



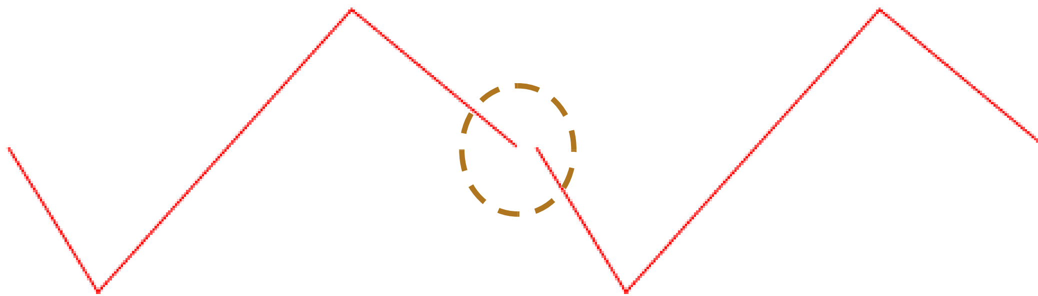
(a) Single maximum



(b) Single maximum and single minimum

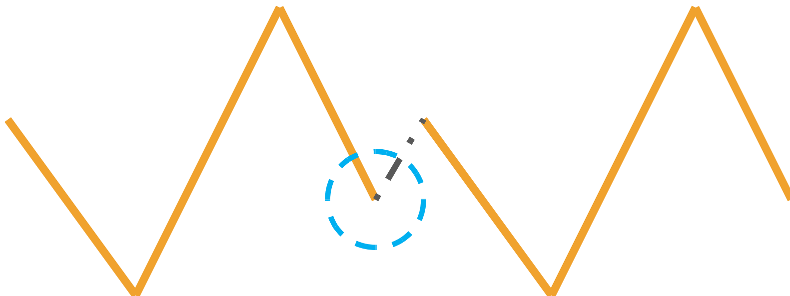
Bitonic Sequence

A *bitonic sequence* is defined as a list with no more than one **LOCAL MAXIMUM** and no more than one **LOCAL MINIMUM**. Endpoints are considered wraparound.



This is ok!
1 Local MAX; 1 Local MIN
The list is bitonic!

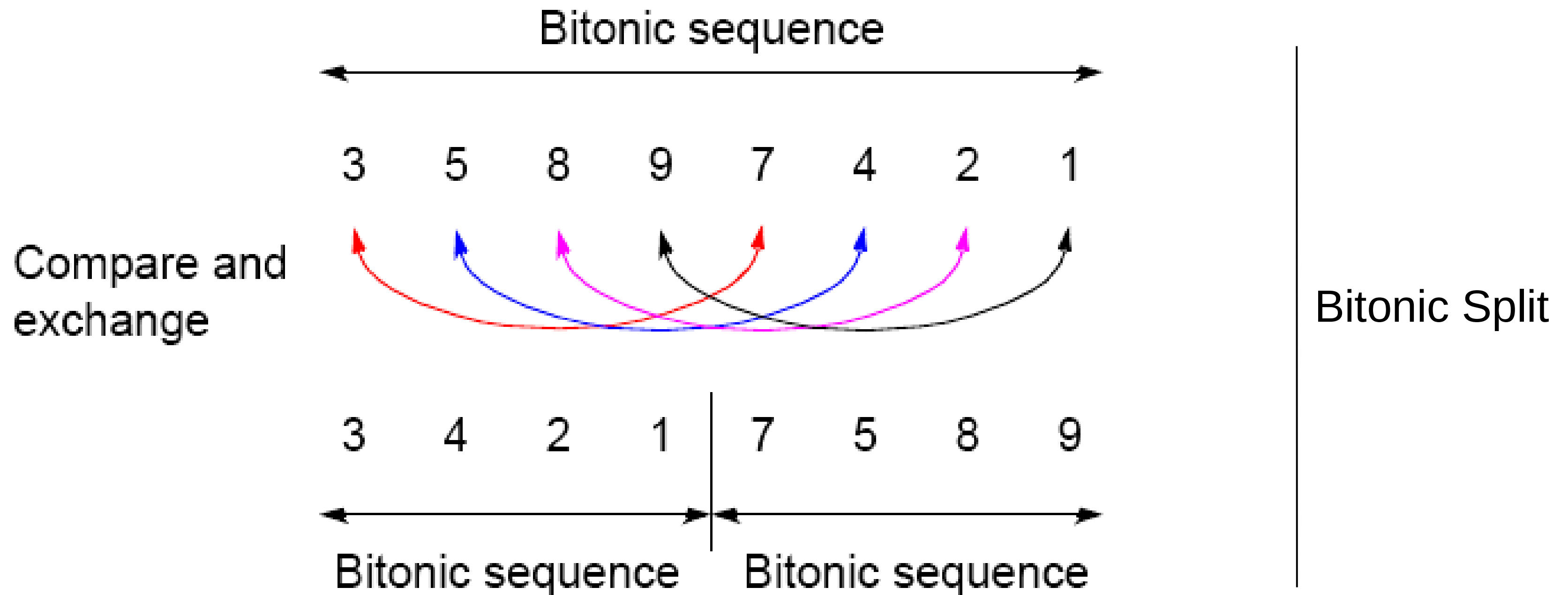
This is **NOT** bitonic!   Why?



1 Local MAX; 2 Local MINs

Properties of Bitonic Sequences

1. Divide a **bitonic list** into two equal halves.
2. Compare-Exchange each item on the first half with the corresponding item in the second half.



Result:
Two bitonic sequences where the numbers in one sequence are <u>all</u> less than <u>all</u> numbers in the other sequence.

Bitonic Merge

Bitonic list:

24 20 15 9 4 2 5 8 | 10 11 12 13 22 30 32 45

Result after Binary-split:

10 11 12 9 4 2 5 8 | 24 20 15 13 22 30 32 45

If you keep applying the BINARY-SPLIT to each half repeatedly, you will get a SORTED LIST !

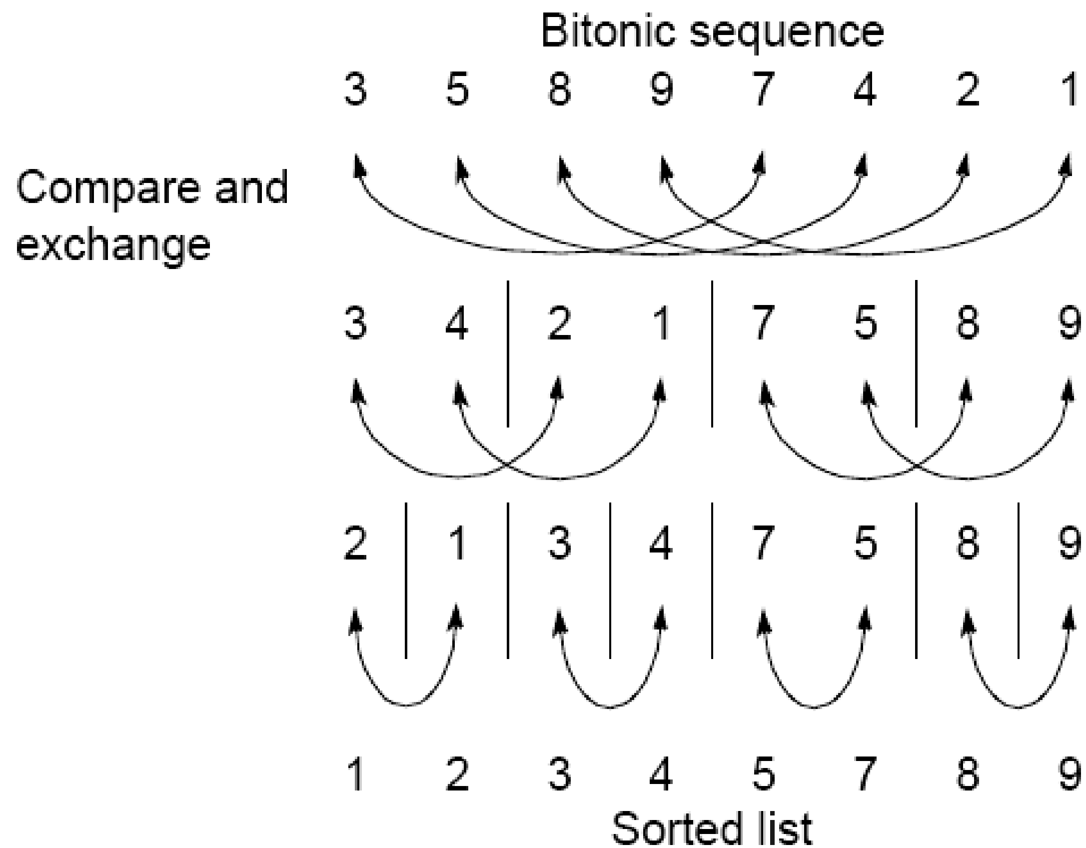
10	11	12	9	4	2	5	8	24	20	15	13	22	30	32	45
4	2	5	8	10	11	12	9	22	20	15	13	24	30	32	45
4	2	5	8	10	9	12	11	15	13	22	20	24	30	32	45
2	4	5	8	9	10	11	12	13	15	20	22	24	30	32	45

Q: How many parallel steps does it take to sort ?

A: **log *n***

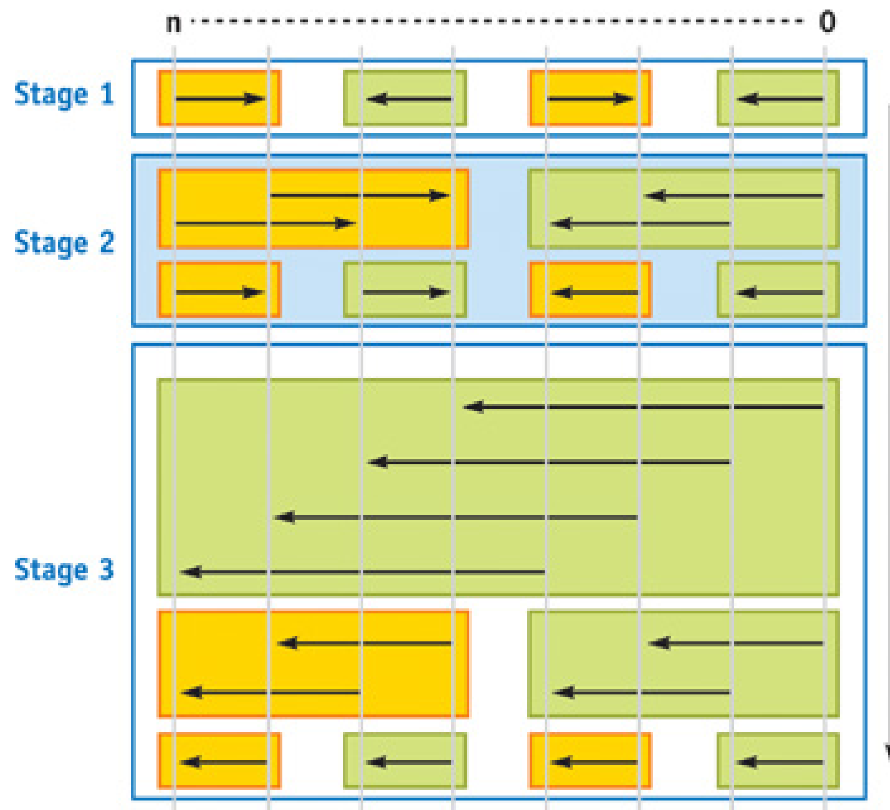
Bitonic Merge

Given a bitonic sequence,
Log n iterations of '*binary split*' will sort the list.



Bitonic Sort

To sort an unordered sequence, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers.
By repeating this process, bitonic sequences of larger and larger lengths obtained until the entire sequence is sorted



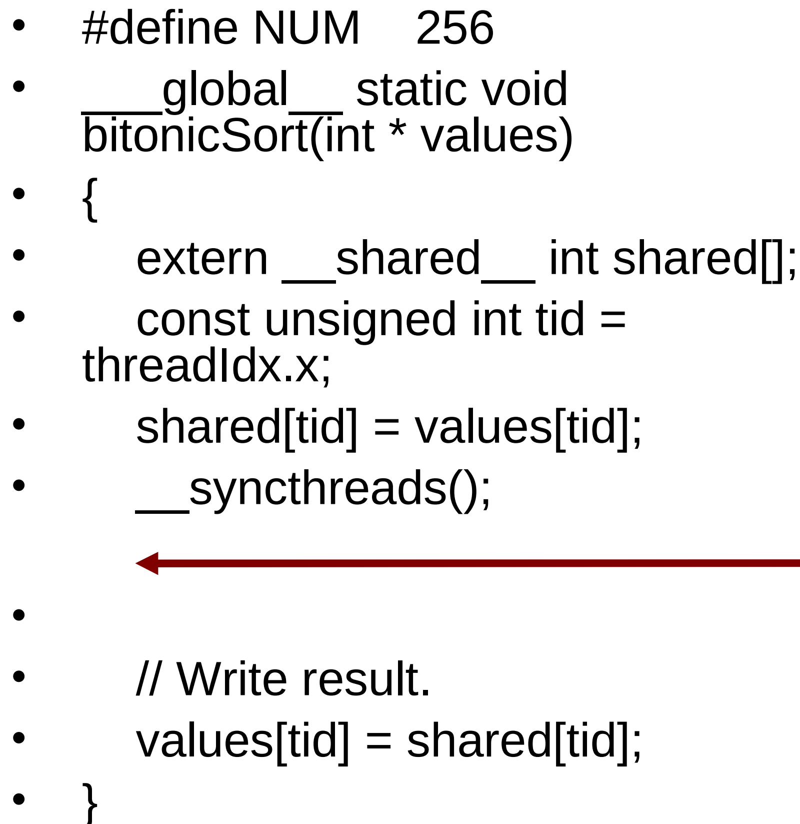

```
for (unsigned int k = 2; k <= NUM; k *= 2){
for (unsigned int j = k / 2; j>0; j /= 2){

Parallel compare/swap


}
}
```

Time: $O(\log^2 n)$

Bitonic Sort on a GPU using CUDA

```
• int main(int argc, char** argv)
• {
•     int values[NUM];
•     for(int i = 0; i < NUM; i++)
•           values[i] = rand();
•     int * dvalues;
•     CUDA_SAFE_CALL(cudaMalloc((void**)&dvalues, sizeof(int) * NUM));
•     CUDA_SAFE_CALL(cudaMemcpy(dvalues, values, sizeof(int) * NUM,
•                                   cudaMemcpyHostToDevice));
•     bitonicSort<<<1, NUM, sizeof(int) * NUM>>>(dvalues);
•     CUDA_SAFE_CALL(cudaMemcpy(values, dvalues, sizeof(int) * NUM,
•                                   cudaMemcpyDeviceToHost));
•     CUDA_SAFE_CALL(cudaFree(dvalues));
•     CUT_EXIT(argc, argv);
• }
```

Bitonic Sort on a GPU using CUDA



```
#define NUM    256
___global__ static void bitonicSort(int * values)
{
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;
    shared[tid] = values[tid];
    __syncthreads();

    // Write result.
    values[tid] = shared[tid];
}
```

```
for (unsigned int k = 2; k <= NUM; k *= 2){
for (unsigned int j = k / 2; j>0; j /= 2){
            unsigned int ixj = tid ^ j;
if (ixj > tid){
                if ((tid & k) == 0)
                    if (shared[tid] > shared[ixj])
                        swap(shared[tid], shared[ixj]);
                else
                     if (shared[tid] < shared[ixj])
                        swap(shared[tid], shared[ixj]);
 }
            __syncthreads();
        }
    }
```