CS213/293 Data Structure and Algorithms 2025

Lecture 18: Sorting

Instructor: Ashutosh Gupta

IITB India

Compile date: November 14, 2025

Sorting

We almost always handle data in a sorted manner.

Computers spend a large percentage of their time in sorting.

We need efficient sorting algorithms.

Many algorithms

There are many sorting algorithms based on various design techniques.

The lower bound of sorting is $\Omega(n \log n)$.

Sorting algorithms

We will discuss the following algorithms for sorting.

- Merge sort
- Quick sort
- Radix sort
- Bucket sort

Topic 18.1

Merge sort

Merge sort : Divide, conquer, and combine

- Divide: Divide the array into two roughly equal parts
- Conquer: Sort each part using mergesort
- Combine: merge the sorted parts

Merge sort

The following code sort array $A[\ell : u-1]$.

Algorithm 18.1: $\textsc{mergeSort}$(int* A, int ℓ, int u)

if $u \leq \ell + 1$ **then** **return**;
$mid := (u + \ell)/2$;
$\textsc{mergeSort}$(A, ℓ, mid);
$\textsc{mergeSort}$(A, mid, u);
$\textsc{merge}$(A, ℓ, p, u);

Merge

Algorithm 18.2: MERGE(int* A, int ℓ, int u, int mid)

```
1  int B[u − ℓ];
2  i := ℓ;
3  j := mid;
4  for  k = 0; k < u − ℓ; k++ do
5      if i < mid and (j ≥ u or A[i] ≤ A[j]) then
6          B[k] := A[i];
7          i := i + 1;
8      else
9          B[k] := A[j];
10         j := j + 1;
11 MEMCOPY( B, A + ℓ, u − ℓ)
```

We cannot do merge in place.

The running time of MERGE is $\Theta(u - \ell)$.

Exercise 18.1

a. Give an optimization that will make MERGE for an ordered array efficient.

b. The condition at line 5 appears to be asymmetric with respect to i and j. Is it?

Example: MERGESORT

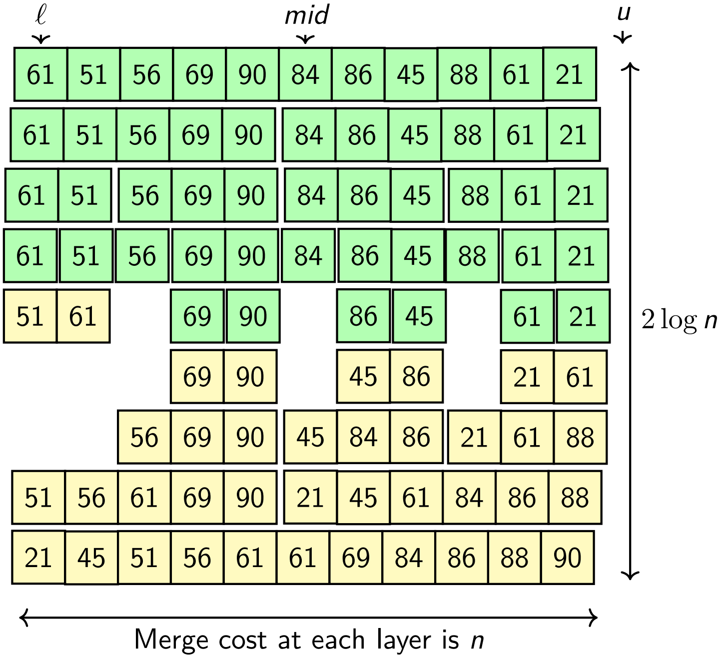
Example 18.1

Consider the following array.

We split the array in the middle and sort the first part.

When partitions are of size one, we start merging. (yellow arrays)

Worst-case complexity: $O(n \log n)$
(visual proof)



Formal proof of the complexity

Let n be the length of the array. Let $T(n)$ be the worst-case complexity of $\textsc{MergeSort}$.

The following recursive equation defines $T(n)$.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + T(n/2) + \Theta(n) & \text{Otherwise.} \end{cases}$$

After a recursive substitution,

$$T(n) = 4T(n/4) + 2\Theta(n/2) + \Theta(n).$$

Formal proof of the complexity(2)

After k recursive substitution,

$$T(n) = 2^k T(n/(2^k)) + k\Theta(n).$$

If $n = 2^k$,

$$T(n) = 2^k T(1) + k\Theta(n).$$

Therefore,

$$T(n) = n + k\Theta(n).$$

Therefore,

$$T(n) \in O(n \log n).$$

Topic 18.2

Quick sort

Quick sort : divide and conquer

- Divide: Partition array in two parts such that elements in the lower part $\leq$ elements in the higher part

- Conquer: recursively sort the parts

Partition

Algorithm 18.3: PARTITION(int* A, int ℓ, int u)

```
pivot := A[ℓ];
i := ℓ − 1;
j := u + 1;
while true do
    do i := i + 1 while pivot > A[i];
    do j := j − 1 while pivot < A[j];
    if i ≥ j then return j ;
    SWAP(A[i], A[j])
```

i is incremented at least once.

The running time of PARTITION is $\Theta(n)$. (Why not $O(n)$?)

Example : partition

Example 18.2

Consider the following input to partition

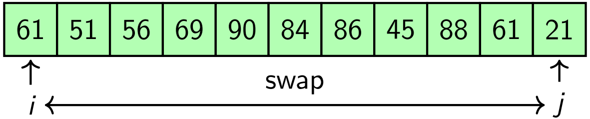
	ℓ $\downarrow$										u $\downarrow$	
	61	51	56	69	90	84	86	45	88	61	21	
$\uparrow$ i												$\uparrow$ j

pivot is 61.

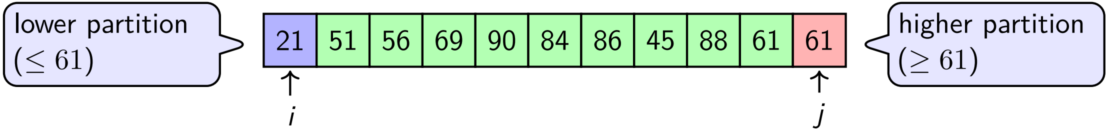
We start scanning the array from both ends, and i and j move inwards.

Example : the first iteration of partition

After the first scan,

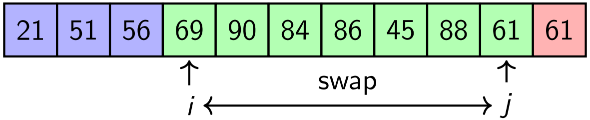


After swap

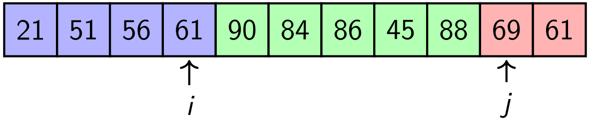


Example : second iteration of partition

After scan

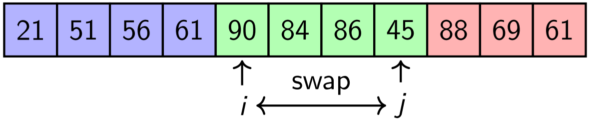


After swap

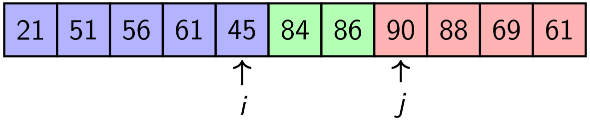


Example : third iteration of partition

After scan



After swap



Example : final iteration of partition

After scan

21	51	56	61	45	84	86	90	88	69	61
				↑	↑					
				j	i					

Since $j \leq i$, the algorithm stops and returns j.

Exercise 18.2

Modify the partition where elements equal to the pivot are moved to the lower partition.

Quick sort

Algorithm 18.4: QUICKSORT(int* A, int ℓ, int u)

if $\ell \geq u$ **then** **return**;
$p :=$ PARTITION(A, ℓ, u);
QUICKSORT(A, ℓ, p);
QUICKSORT(A, $p + 1$, u);

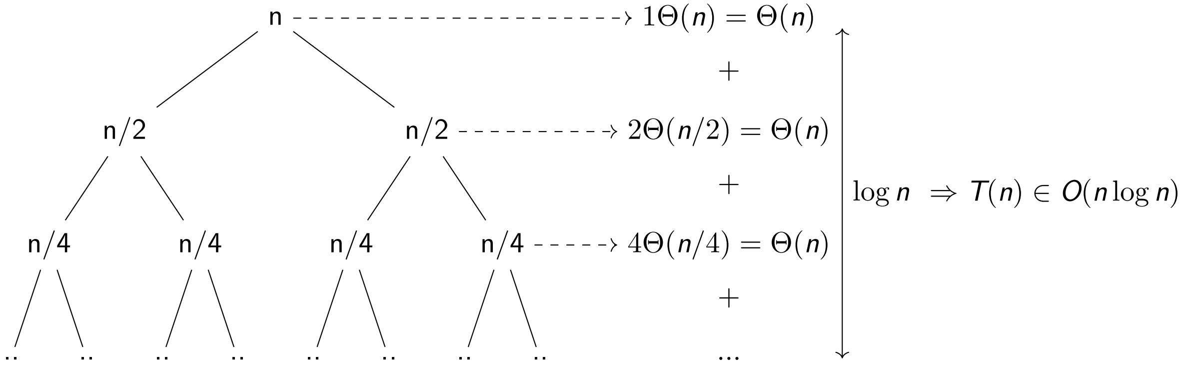
Topic 18.3

Analysis of quick sort

Best case execution

Every time the array splits into two halves.

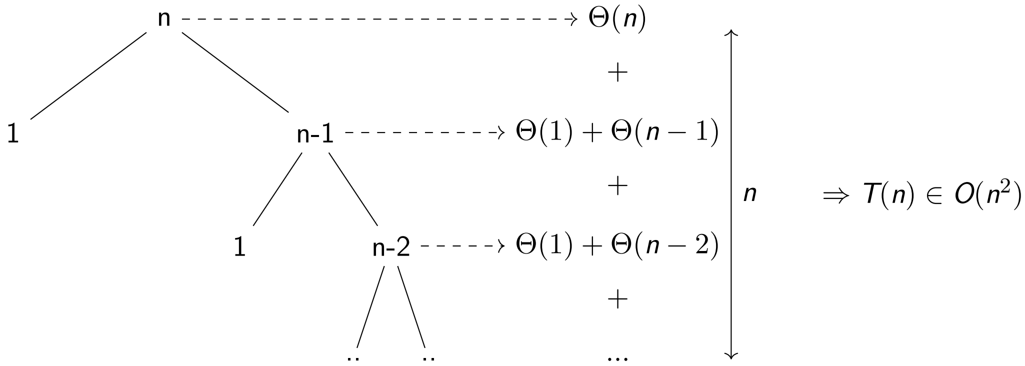
$$T(n) = 2T(n/2) + \Theta(n)$$



Worst case execution

Every time the array splits into an array of size 1 and the rest.

$$T(n) = T(1) + T(n-1) + \Theta(n)$$



When does the worst case occur?

- Worst case occurs on sorted or reverse sorted array. ¯_(ツ)_/¯

Exercise 18.3

Can we modify quick sort such that on a sorted array its running time is $O(n)$?

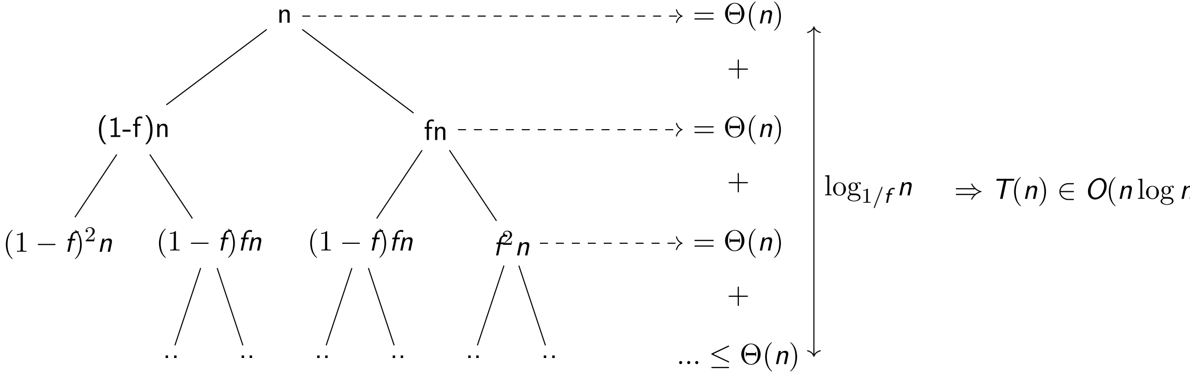
Properties of quick sort

- In-place sorting
- practical, average $O(n \log n)$(with small constants), but worst $O(n^2)$

Other fractional splits

Every time the array splits into fn and $(1-f)n$ arrays, where $f > 1/2$.

$$T(n) = T(fn) + T((1-f)n) + \Theta(n)$$



Topic 18.4

Randomized quicksort

How to get to the average case?

- Partition around the "middle" element?
- Partition around the random element.

Randomized quicksort

Algorithm 18.5: RANDOMQUICKSORT(int* A, int ℓ, int u)

```
if ℓ ≥ u then return;
i := RANDOM(ℓ, u);
SWAP(A[i], A[ℓ]);
p := PARTITION( A, ℓ, u );
QUICKSORT( A, ℓ, p );
QUICKSORT( A, p + 1, u );
```

Analyzing randomized quicksort

Assume all elements are distinct.

Due to partition around a random element, all splits are equally likely.

Analyzing Randomized quicksort

Definition 18.1

Let $T(n)$ be the expected number of comparisons needed to quicksort n numbers.

Since each split occurs with probability $1/n$, $T(n)$ has value $T(i-1) + T(n-i) + n - 1$ with probability $1/n$.

$$T(n) = \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i) + n - 1) = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n - 1$$

Solving the recurrence relation

We have seen the recurrence relation before in the inserting permutations of numbers in BST.

We had proven $T(n) \in O(n \log n)$

What does expected running time mean?

- Algorithm may have different running time at different times
- Running times are input-independent. It depends on random number generators.

Topic 18.5

Radix sort

Radix partition

Sort the array according to the bth bit.

Algorithm 18.6: RadixPartition(int* A, int b, int ℓ, int u)

$i := \ell - 1$; $j := u + 1$;
while *true* **do**
  do $i := i + 1$ **while** $2^b \& A[i] = 0$ *and* $i \leq u$;
  do $j := j - 1$ **while** $2^b \& A[j] \neq 0$ *and* $j \geq \ell$;
  if $i \geq j$ **then** **return** j ;
  SWAP$(A[i], A[j])$

Example: radix partition

Example 18.3

Consider the following numbers.

- $61 = 0111101$
- $51 = 0110011$
- $56 = 0111000$
- $69 = 1000101$
- $90 = 1011010$
- $84 = 1010100$
- $86 = 1010110$
- $45 = 0101101$
- $88 = 1011000$
- $61 = 0111101$
- $21 = 0010101$

radix partition divides the numbers as follows.

- $61 = 0111101$
- $51 = 0110011$
- $56 = 0111000$
- $45 = 0101101$
- $61 = 0111101$
- $21 = 0010101$
- $69 = 1000101$
- $90 = 1011010$
- $84 = 1010100$
- $86 = 1010110$
- $88 = 1011000$

RadixSort

We start from the most significant bit (leftmost bit).

Algorithm 18.7: $\textsc{RadixSort}($ int* A, int ℓ, int u $)$

Let b be the number of bits needed for the largest number in A;
$\textsc{RadixSortRec}$(A, b, ℓ, u);

Algorithm 18.8: $\textsc{RadixSortRec}($ int* A, int b, int ℓ, int u $)$

if $\ell \geq u$ *or* $0 > b$ **then** **return**;
$p :=$ $\textsc{RadixPartition}($ A, b, ℓ, u $)$;
$\textsc{RadixSortRec}($ A, $b-1$,ℓ, p $)$;
$\textsc{RadixSortRec}($ A, $b-1$,$p+1$, u $)$;

Worst-case run time complexity: $O(bn)$

Exercise 18.4

Is this complexity better than $O(n \log n)$?

Topic 18.6

Bucket sort

Bucket sort

Radix sort does well when $b < \log n$, which implies low variation and high repetition among keys.

Let us suppose that we know that the numbers in the array are from set $\{0, ..., m-1\}$.

We can create m buckets.

We scan the array from one end to another and place the number in their respective buckets, therefore sorting the array.

Exercise 18.5

Give a situation when bucket sort will be the best sorting algorithm?

Topic 18.7

Why $O(n \log n)$?

Comparison sort

A sorting algorithm takes an input sequence and produces a permutation of the input.

Definition 18.2

A comparison sort determines the sorted order only based on comparisons between the input elements.

Let us assume our input has only distinct elements. All we need to do is strict comparison.

Decision tree model of executions.

Imagine a comparison sort algorithm, that needs to identify the sorting permutation.

It executes a sequence of comparisons over the elements of inputs.

Example: Comparison sort

Let us consider BUBBLESORT, which is an example comparison sort.

Algorithm 18.9: BUBBLESORT(int* A, int n)

for $i = 0; i < n - 1; i++$ **do**
  for $j = 0; j < n - i - 1; j++$ **do**
    if $A[j] > A[j+1]$ **then**
      SWAP(A, j, j+1);

To understand the behavior or the algorithm, let us suppose $A = [a_0, a_1, a_2]$.

We will consider all possible executions of the algorithm and create an execution tree.

Exercise 18.6

How many possible different paths a run of BUBBLESORT can take for the input array of size 3?

Commentary: An execution is the run of the program on a given input. An execution tree is the representation of the all possible executions of the program on all possible inputs. The representation looks like a tree because of conditions in the program that may take the program along one path or another.

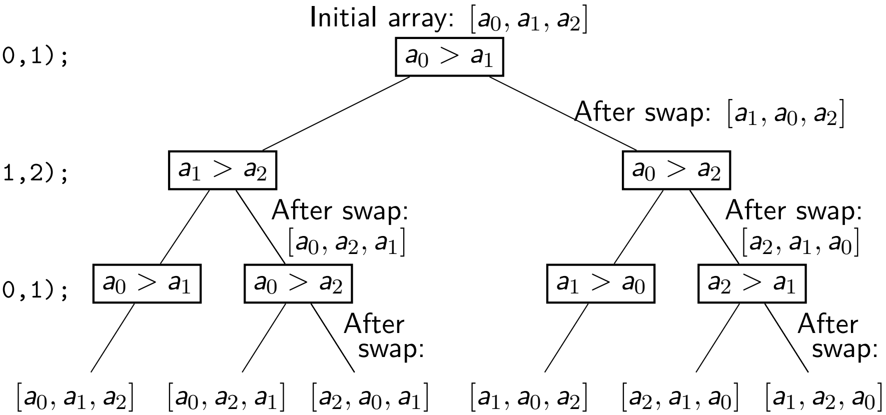
Example: Execution tree of BUBBLESORT

Now let us consider all possible executions of BUBBLESORT. We compare elements of the array. If the condition holds(right child), we run swap.

Unrolled BUBBLESORT

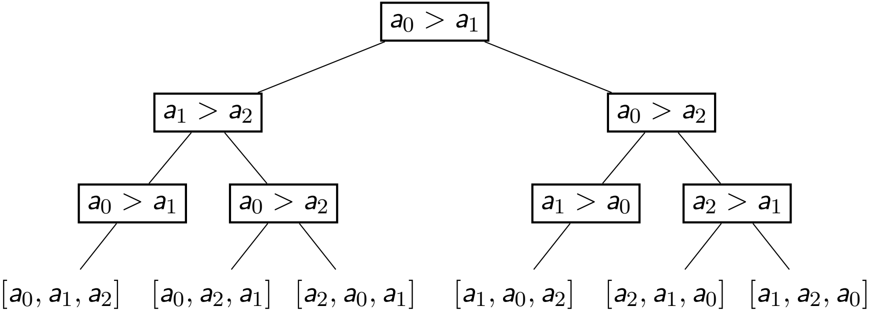
```
if(A[0]>A[1]) swap(A,0,1);

if(A[1]>A[2]) swap(A,1,2);

if(A[0]>A[1]) swap(A,0,1);
```

Execution tree



Structure of the execution tree of any possible comparison sort

Let us ignore the array updates and the intermediate states. In the tree, we only track the elements (not positions) compared and the final permutation.



We can create the abstraction of the execution tree of any comparison sort algorithm.

The above is a binary tree that has $n!$ leaves.

Best worst-case performance

Theorem 18.1
Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Proof.
An execution tree has $n!$ leaves.

The minimum height of the tree is $O(\log(n!))$, which is $O(n \log n)$. Hence proved! $\square$

Topic 18.8

Sorting in C++

IntroSort

Algorithm 18.10: INTROSORT(Array A, int maxDepth=2*$\lfloor \log_2(len(A)) \rfloor$)

```
if len(A) < 16 then
    INSERTIONSORT(A);
else
    if maxdepth > 0 then
        // Quick sort step;
        p = partition(A); // Select pivot and partition
        INTROSORT(A[1:p-1],maxdepth-1);
        INTROSORT(A[p+1:n],maxdepth-1);
    else
        HEAPSORT(A);
```

Exercise 18.7

a. What is the default algorithm in the sort command of GNU core utils?

b. Rust is planning to replace GNU core utils. Their sort is six times faster (How?)

IntroSort

Topic 18.9

Tutorial Problems

Exercise: sorting a sorted array

Exercise 18.8

a. Modify the PARTITION such that it detects that the array is already sorted.

b. Can we use this modification to improve quicksort?

Exercise: straight radix sort

Exercise 18.9

Modify RadixSort such that it considers bits from right to left.

Commentary: We need to ensure that later iterations of sorting do not disrupt earlier work.

Exercise: Execution tree for Heap sort

Exercise 18.10

Draw the execution tree for the Heap sort. Let us suppose the input array is of size 3.

Exercise: quickselect (Final 2023)

Exercise 18.11

The quickselect algorithm finds the kth smallest element in an array of n elements in average-case time O(n). Let k be a fixed number. The algorithm is as follows:

Algorithm 18.11: QUICKSELECT(Array A, int k)

$p := random(1, n)$
$swap(A, 1, p)$
$p := partition(A)$ // Usual quick sort partition
if $p = k$ **then** **return** A[p] ;
if $p > k$ **then**
  return QUICKSELECT(A[1 . . . p-1], k)
else
  return QUICKSELECT(A[p+1 . . . n], k-p)

1. Suppose the pivot was always chosen to be the first element of A. Show that the worstcase running time of quickselect is then $O(n^2)$.
2. Show that the expected running time of (randomized) quickselect is $O(n)$.

Topic 18.10

Problem

True or False

Exercise 18.12

Mark the following statements True / False and also provide justification.

1. Merge sort is a stable sorting algorithm, meaning it maintains the relative order of equal elements in the sorted output.

Exercise: In-place sorting

Exercise 18.13

Give the sorting algorithms that are not in-place sorting algorithms. An algorithm is in-place sorting algorithm if does not use more than $O(1)$ extra space and update is only via replace or swap.

Exercise: Infeasible paths in Bubble sort (Quiz 2024)

Exercise 18.14

Consider the following bubble sort algorithm, where A is the pointer to the input array and n is the length of the array. Let us suppose A contains distinct elements. The function call to SWAP(A, j, j+1) exchanges the values of $A[j]$ and $A[j+1]$.

Algorithm 18.12: BUBBLESORT(int* A, int n)

for $i = 0; i < n - 1; i++$ **do**
    for $j = 0; j < n - i - 1; j++$ **do**
        if $A[j] > A[j+1]$ **then**
            SWAP(A, j, j+1);

Let us suppose $n = 3$. The **if** condition in the BubbleSort will execute three times as follows.

Algorithm 18.13: BUBBLESORTUNROLLED(int* A, int $n = 3$)

if $A[0] > A[1]$ **then** SWAP(A, 0, 1) ;
if $A[1] > A[2]$ **then** SWAP(A, 1, 2) ;
if $A[0] > A[1]$ **then** SWAP(A, 0, 1) ;

In the above unrolled program, each *if* condition can either be true or false. Each combination is a path of the program. Therefore, there are 8 paths in the program. A path is feasible if each branch condition along the path is true if the branch is taken otherwise it is false. Some paths are infeasible. For example, consider a path where the first two conditions are false and the last one is true. This path is infeasible because the first and the last conditions are the same and the content of A is not changed between the two conditions.

1. Give one more path that is not feasible for $n = 3$.
2. Give expression of the number of infeasible paths in the BubbleSort in terms of n?
3. For $n = 4$, how many paths are there in the BubbleSort and how many infeasible paths are there in the BubbleSort?

Exercise: Sorting networks (Final 2024)

Exercise 18.15

Consider the following sorting primitive:

```
SortTwo(A,i,j) { if( A[i] > A[j] ) swap(A, i, j); }
```

For a given n and A, an application of a sequence of the primitive is called sorting network. For example, the sequence `SortTwo(A,0,1); SortTwo(A,1,2); SortTwo(A,0,1);` is a sorting network, which successfully sorts an array A of size 3. Prove that if a sorting network correctly sorts every sequence of 0's and 1's for a given length, then it correctly sorts every arbitrary sequence of integers of the same length.

Commentary: The above statement is called 0-1 principle. `https://hwlang.de/algorithmen/sortieren/networks/nulleinsen.htm`

End of Lecture 18