Selecting and Sorting

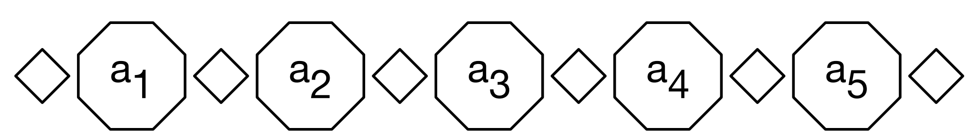
Thomas Schwarz, SJ

Permutations

- A permutation of the set $\{1,2,\ldots,n\}$ is a reordering of the numbers where each number between 1 and *n* appears exactly once.

Permutations

- How many permutations are there?
  - Use recurrence!
    - In a permutation of $\{1,2,\ldots,n\}$, where is the n located?
    - There are $n-1$ other numbers.
    - This gives us $n-2$ gaps and spots before and after



Permutations

- Let $n!$ be the number of permutations of n elements
  - This gives us the recurrence
    - $n! = n \cdot (n-1)!$
  - which can be unfolded very simply
    - $$n! = \prod_{i=1}^{n} i$$

Permutations

How do we determine its asymptotic growth?

$$n! = \prod_{i=1}^{n} i$$

Use Logarithms!

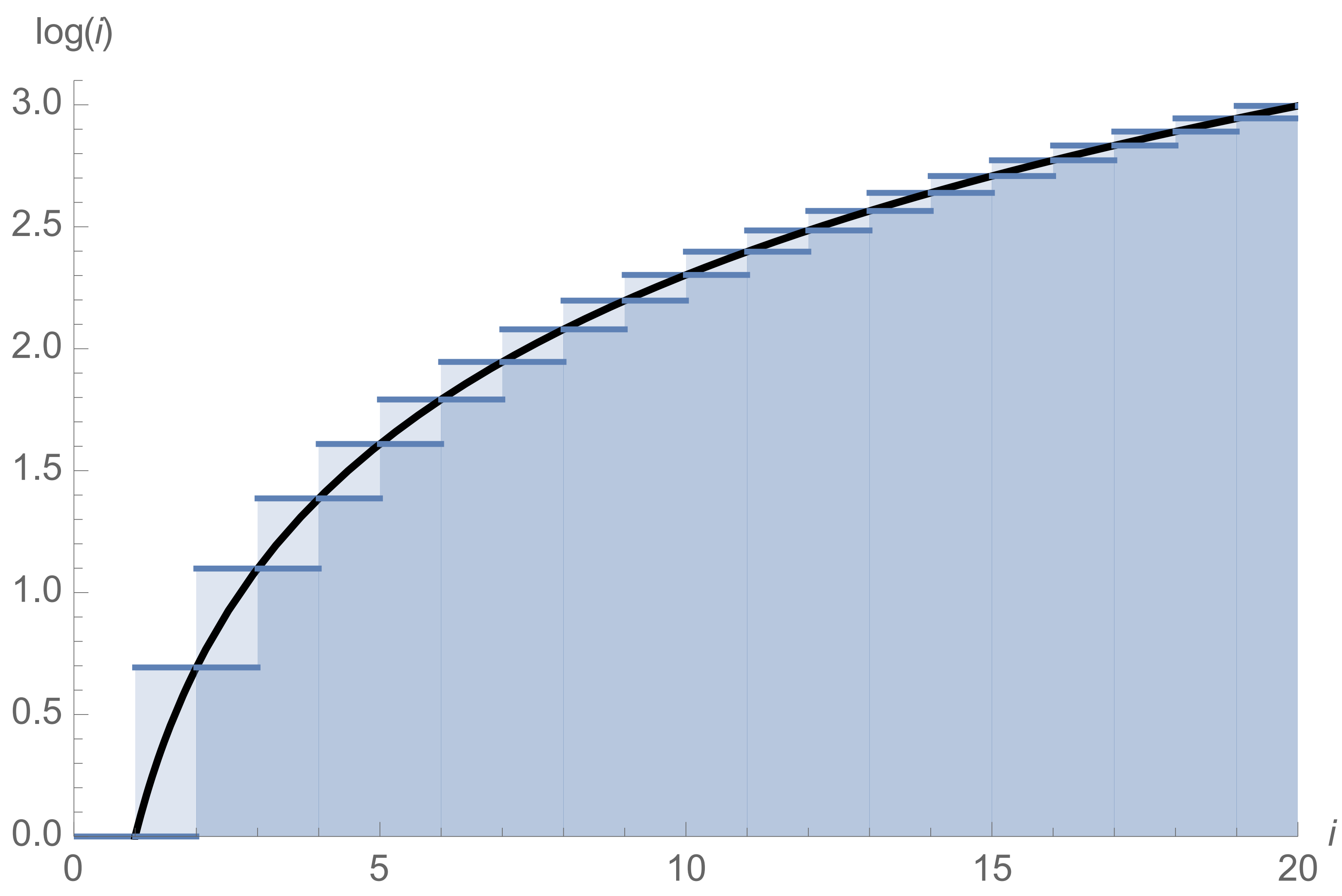
Permutations

- Approximation of the factorial

  Use $\log n! = \sum_{i=1}^{n} \log(i)$

Use an integral!

Permutations



$$\sum_{i=1}^{n-1} \log(i) < \int_1^n \log(x)dx < \sum_{i=1}^{n} \log i$$

Permutations

$$\log(n!) = \sum_{i=1}^{n} \log(i)$$

$$\approx \int_{i=1}^{n} \log(x)dx$$

$$= [x \log x - x]_1^n$$

$$= n \log(n) - n + 1$$

Permutations

Therefore

$$\begin{aligned} n! &\approx \exp(n\log(n) - n - 1) \\ &= \exp(\log(n^n) - n + 1) \\ &= n^n \cdot e^{-n} \cdot e \\ &= e \cdot \left(\frac{n}{e}\right)^n \end{aligned}$$

Permutations

An analysis of the error substituting the Riemann sum for an integral gives Stirling's formula (invented by de Moivre)

$$\sqrt{2\pi}n^{n+\frac{1}{2}}e^{-n} \leq n! \leq en^{n+\frac{1}{2}}e^{-n}$$

Simple Sorting Algorithms

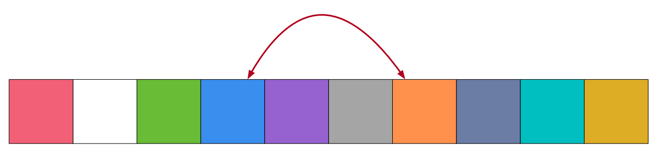
Thomas Schwarz, SJ

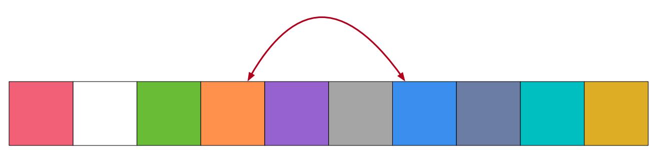
Simple Sorting

- Sorting algorithms can be in-place:
  - No additional memory is needed
  - Sorting algorithms can be based on swaps

Simple Sorting

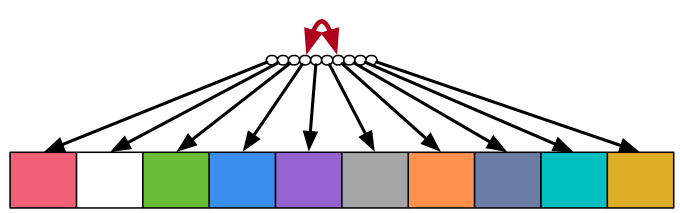
- Implementing in-place sorting with swaps
  - Do not move large objects:

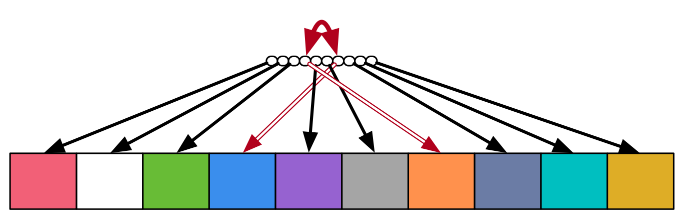


- 

```
temp = blue.copy()
blue = orange.copy()
orange = temp.copy()
```

  - Instead move pointers to objects: (also more natural in Python)



- 

```
arr[3],arr[6] = arr[6],arr[3]
```

Bubble Sort

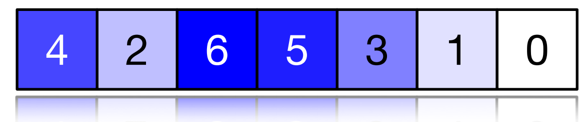
Thomas Schwarz, SJ

Bubble Sort

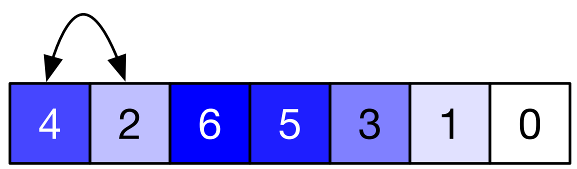
- Idea of bubble sort:
  - Repeatedly swap adjacent elements in an array until they are in order
    - Reminder: Swaps in Python are easy:
      - `arr[i],arr[i+1] = arr[i+1],arr[i]`
- ```
  while not done:
      for i in range(len(arr)-1):
          if arr[i] > arr[i+1]:
              arr[i],arr[i+1]=arr[i+1],arr[i]
  ```
  ```

Bubble Sort

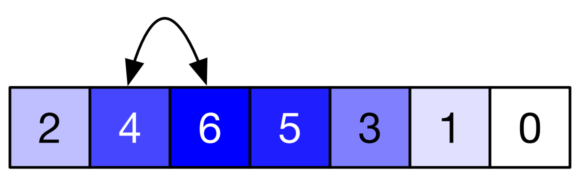
- Example: Sort



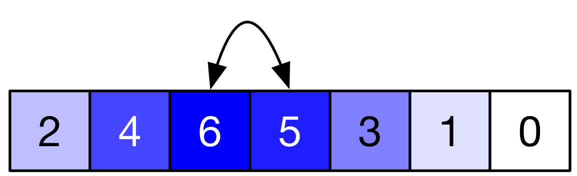
  - First pass: Check first pair



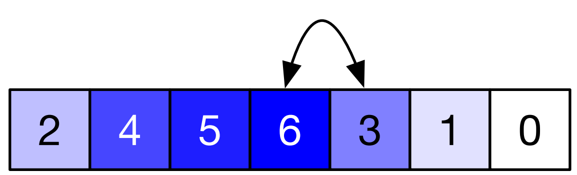
    - Swap and move on

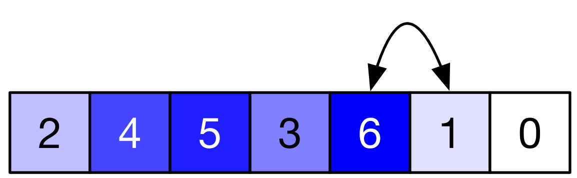


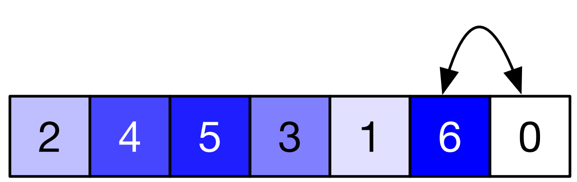
    - No swap necessary, move on



Bubble Sort

- Example:
  - Swap and move on
    - 

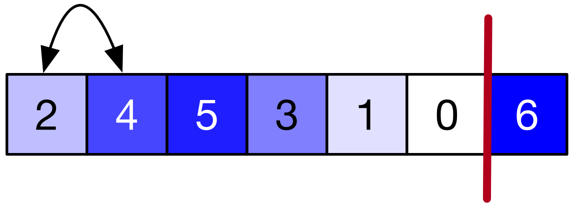
  - Swap and move on
    - 

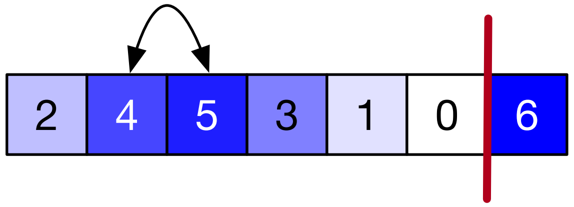
  - Swap and move on
    - 

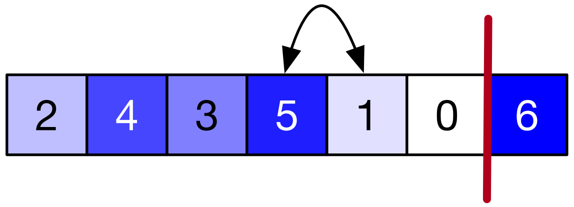
Bubble Sort

- Example:
  - Swap and move on
2	4	5	3	1	0	6
  - Array is still not sorted, so we need to continue
  - However: Notice that the maximum element has been picked up and is now at the correct position
  - We only have to order the first $n - 1$ positions

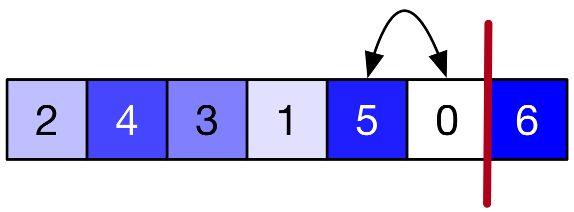
Bubble Sort

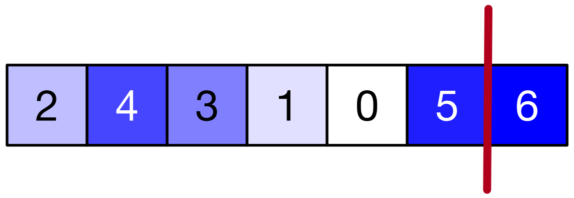
- Example
  - Second pass:
    - 

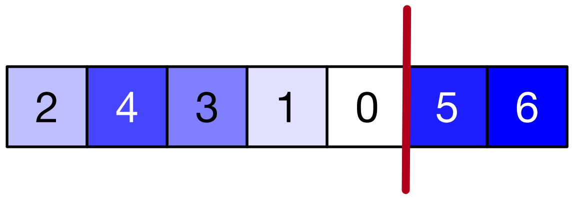
    - 

    - 

Bubble Sort

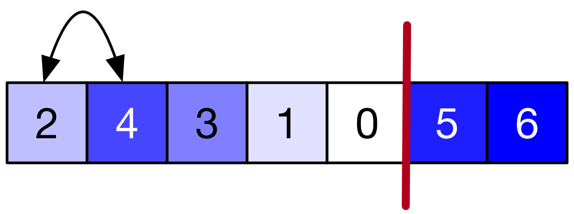
- Example (Second Pass):
  - 

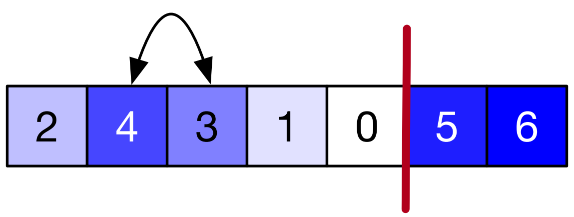
  - 

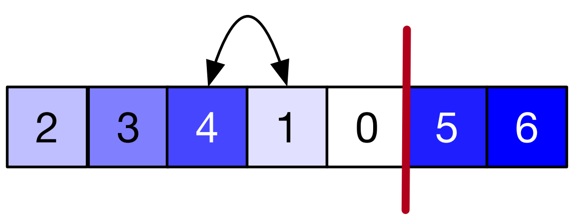
- The maximum in the remaining array has now reached its correct point
  - 

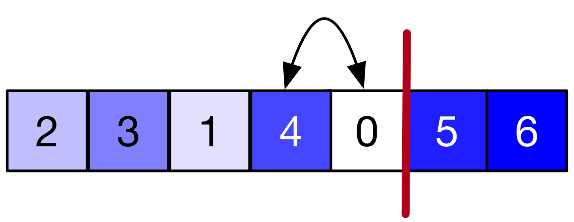
Bubble Sort

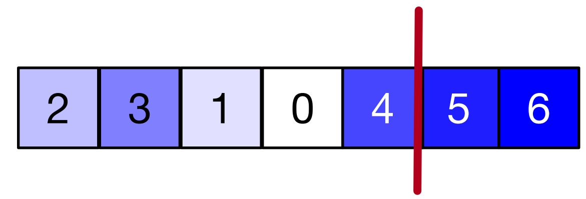
- Example: Third Pass

- 

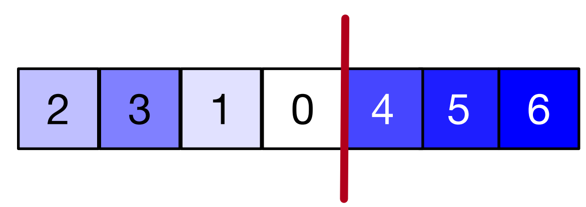
- 

- 

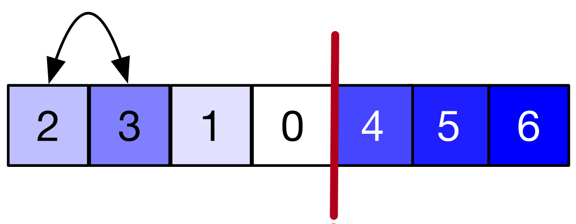
- 

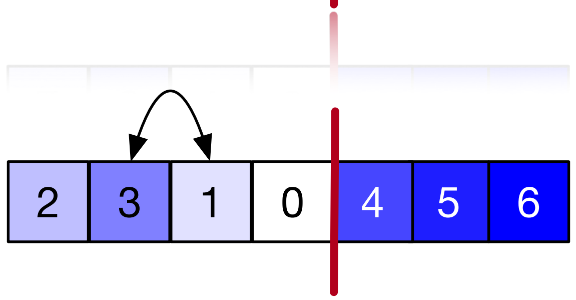
- 

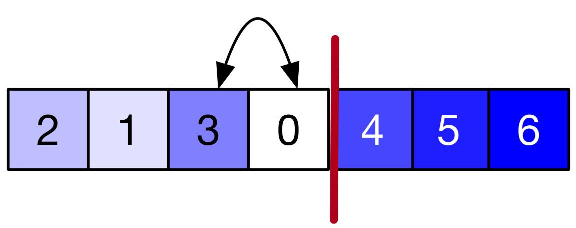
- Third largest element has bubbled up to the correct place

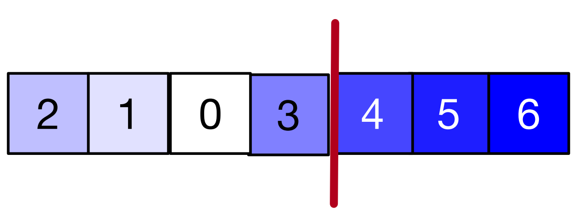
- 

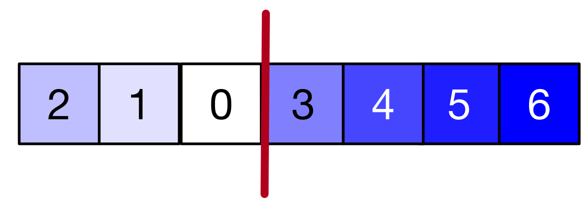
Bubble Sort

- Fourth pass
  - 

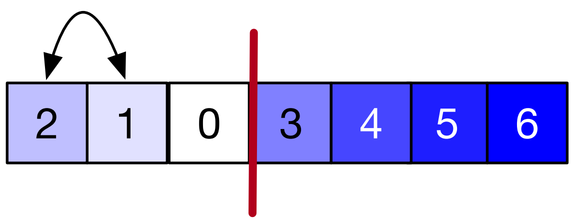
  - 

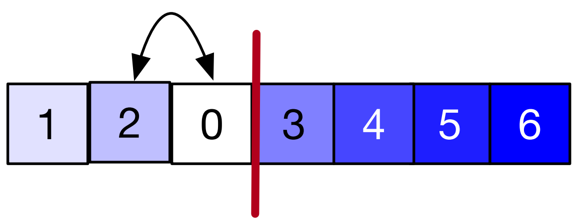
  - 

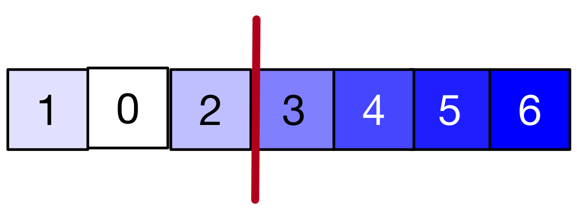
  - 

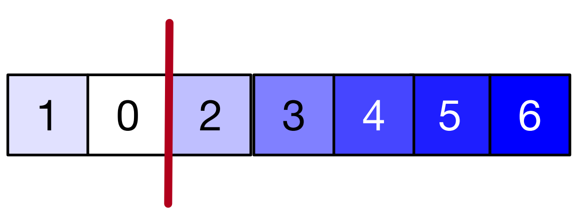
- Now 3 has bubbled up
  - 

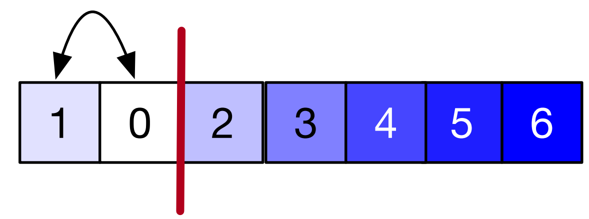
Bubble Sort

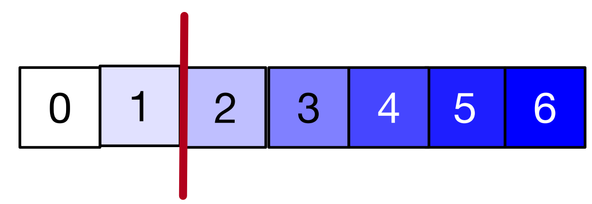
- Fifth Pass
  - 

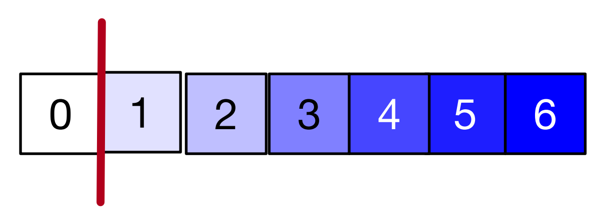
  - 

  - 

- 2 has bubbled up
  - 

- Final Pass
  - 

  - 

- 1 has bubbled up, and a singleton is always sorted:
  - 

Bubble Sort

- We need one less pass than there are array elements
  - And we do not need to look at the last elements of the array

```
def bubblesort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j],arr[j+1]=arr[j+1],arr[j]
```

Bubble Sort

- Potential improvements:
    - After each pass, the elements after the last swap are already in order
        - We can skip the corresponding passes
            - But need to keep track of the last swap

Bubble Sort

- Performance:
  - At pass $i, i = 0,1,\ldots,n-2$, we compare $n-i-1$ values
  - This means, we make
    - $(n-1)+(n-2)+\ldots+2+1=\dfrac{n(n-1)}{2}$ comparisons

Bubble Sort

- If we use the last swap trick:
  - Best case behavior: The array is sorted, we did not do any swap, and we are done after a single pass with $n-1$ comparisons

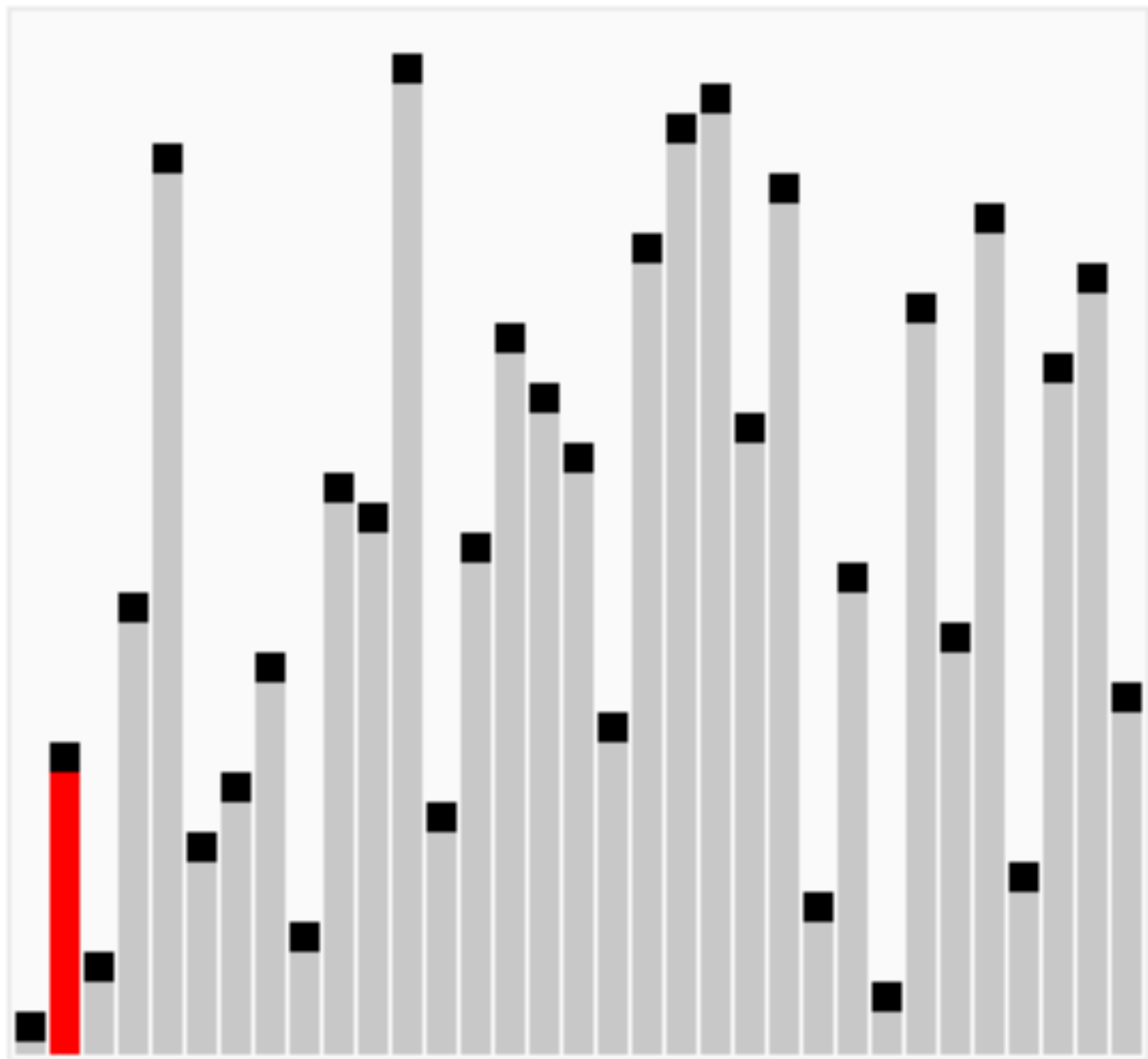
Bubble Sort

- Bubble sort is known to be the least efficient sort for data that is not already sorted
    - Among the sorting algorithms that do not try to be horrible

Cocktail Sort

- Bubble sort will move small elements only slowly to their correct position
  - Cocktail sort makes one pass from the left to the right
    - Moves maximum to its rightful spot
  - Then the next pass from the right to the left
    - Moves minimum to its rightful spot
  - Then the next pass from left to the right starting with second element and ending before the last one

Cocktail Sort



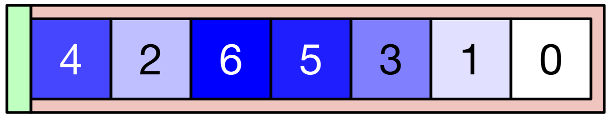
Insertion Sort

Thomas Schwarz, SJ

Insertion Sort

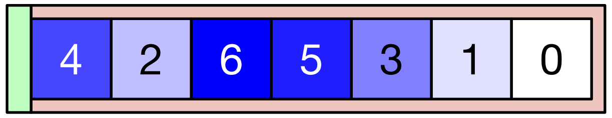
- Idea:
  - Break the array into a sorted and an unsorted part
    - Move first element of the unsorted part into the correct position in the sorted array

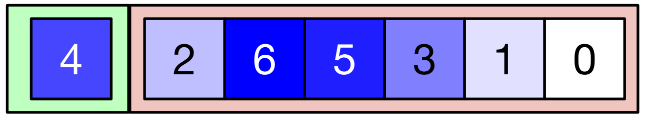
Insertion Sort

- Example:
  - Sort 

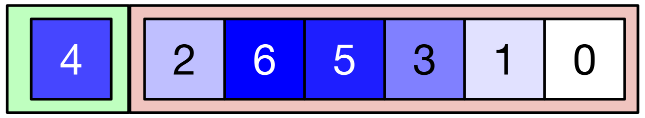
    - Reddish part is unsorted: initially whole array
    - Greenish part is sorted: initially empty

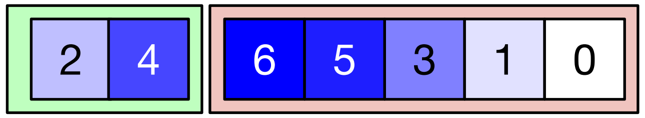
Insertion Sort

- Example:
  - 

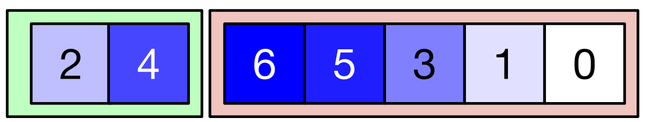
  - First element in the red part is 4:
  - Insert 4 into the green part
  - 

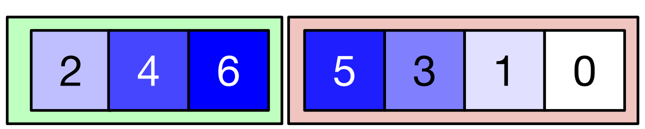
Insertion Sort

- Example
  - 

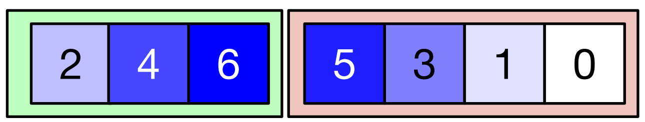
  - Next unsorted element is 2
  - Compare with 4
  - Insert in front of 4
  - 

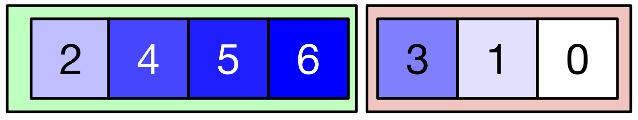
Insertion Sort

- Example
  - 

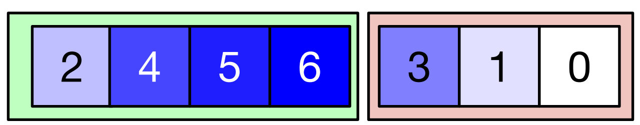
  - Next unsorted element is 6
    - Compare with 2, then 4
    - Insert after 4
  - 

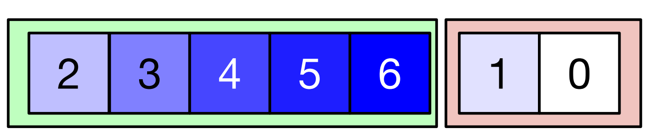
Insertion Sort

- Example
  - 

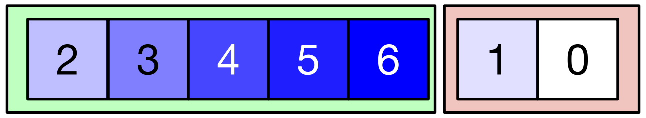
  - Next unsorted element is 5
  - Compare with 2, 4, 6
    - Insert before 6
  - 

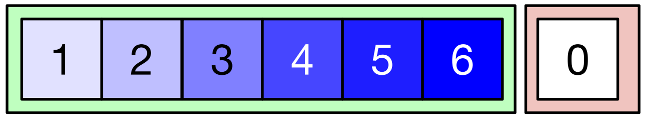
Insertion Sort

- Example
    - 

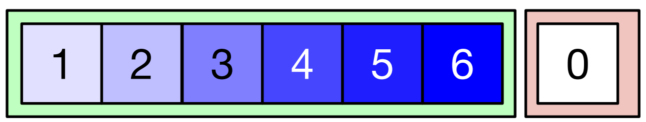
    - Next unsorted element is 3
    - Compare with 2, then 4
    - Insert before 4
    - 

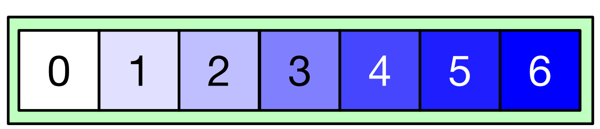
Insertion Sort

- Example
  - 

  - Next comes 1
  - Compare with 2
  - Insert before 2
  - 

Insertion Sort

- Example
  - 

    - Final unsorted element is 0
    - Compare with 1
    - Insert before 1
  - 

  - We are done

Insertion Sort

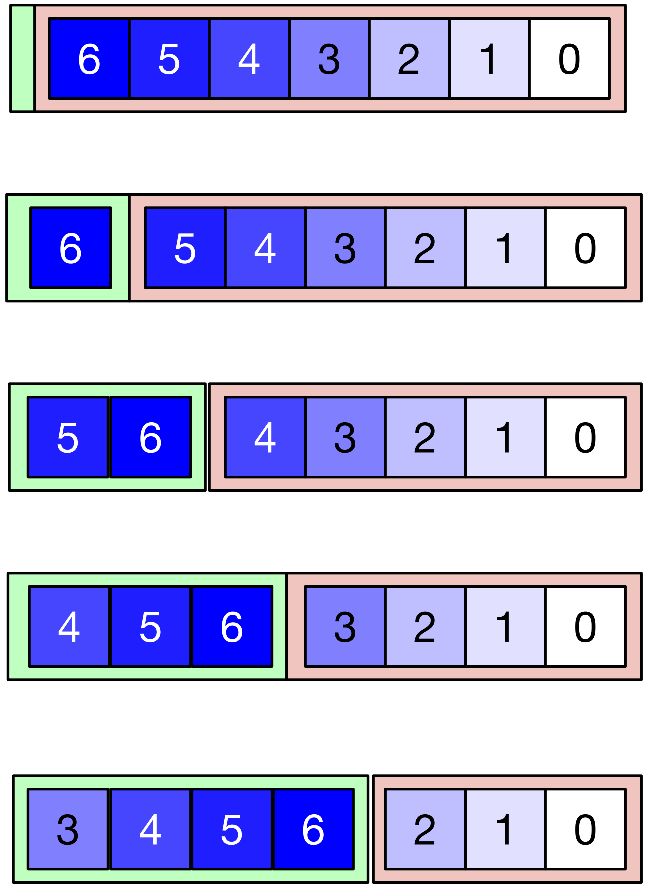
- Performance:
  - Inserting at a specific index in an array means moving the elements after the insertion
    - This is a big hidden cost
  - Inserting at a specific index into a linked list only involves finding the insertion point and constant link resetting work
  - However, we can now avoid comparisons
  - To insert into a sorted array of length i
    - only need on average $\frac{i}{2}+1$ comparisons

Insertion Sort

- Average case:
  - Pass i has $1 + \frac{i}{2}$ comparisons
  - Total of $\sum_{i=0}^{n-1}(1 + \frac{i}{2}) = n + \frac{1}{2}\frac{n(n-1)}{2}$ comparisons

Insertion Sort

- Best Case:
  - Only one comparison per pass:
    - New element inserted into the sorted part is smaller than the current minimum of the part
  - Original array is ordered from maximum to minimum



Selection

Selection Problems

- Given an unordered array:
  - Find the k-largest (-smallest) element in an unordered array
  - Naïve Solution:
    - Sort (usually in time $\Theta(n \log n)$)
    - Pick element $n - k$ or k of the sorted array

Selection Problem

- Finding the maximum
- Finding the maximum and minimum at the same time
- Finding the k^{th} largest element
- Finding the median

Maximum

- Obvious algorithm:

```
def max(array):
    result = array[0]
    for i in range(1, len(array)):
        if array[i]>result:
            result = array[i]
    return result
```

- $n - 1$ comparisons

Maximum

- Toy algorithm:
  - Partition array into $\lfloor n/2 \rfloor$ pairs.
    - (There might be an additional element).
  - Use one comparison in order to select the largest of each pair (plus the odd one out if exists)
  - These form an array of length $\lfloor n/2 \rfloor + 1$
  - Recursively call the toy algorithm

Maximum

- What is the recurrence relation?

Maximum

- $T(n) = T(n - \lfloor n/2 \rfloor) + \lfloor n/2 \rfloor$
- T(2) = 1

- Now use substitution to get an idea of solving the recurrence

Maximum

- Assume n is a power of 2

Maximum

- Recurrence then becomes
  - $T(n) = T(n/2) + n/2, \quad T(2) = 1$
  - $\qquad = T(n/4) + n/4 + n/2$
  - $\qquad = T(n/8) + n/8 + n/4 + n/2$
  - $\qquad \ldots$
  - $\qquad = T(2) + 2 + 4 + 8 + \ldots + n/8 + n/4 + n/2$
  - $\qquad = n - 1$

Maximum

- Now prove by induction for all $n \in \mathbb{N}$
- $T(n) = T(n - \lfloor n/2 \rfloor) + \lfloor n/2 \rfloor$
- $T(2) = 1$

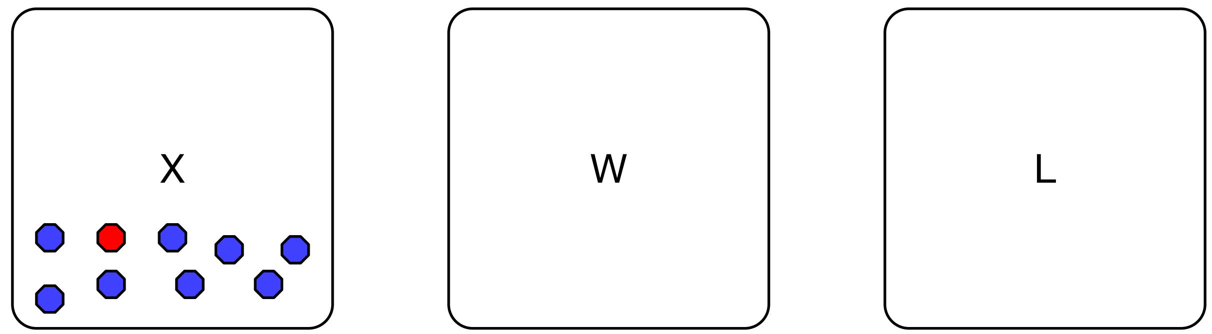
Maximum

- Induction Hypothesis: $T(m) = m - 1$ if $m < n$.
- $T(n)$
  - $= T(n - \lfloor n/2 \rfloor) + \lfloor n/2 \rfloor$
  - $= n - \lfloor n/2 \rfloor - 1 + \lfloor n/2 \rfloor$
  - $= n - 1$

Maximum

- In fact:
  - *Theorem: Finding the maximum of an array of length n costs at least $n-1$ comparisons*
  - *Proof*: Place all elements into three buckets:
    - One for not-looked at
    - One for won all comparisons
    - One for lost at least one comparison

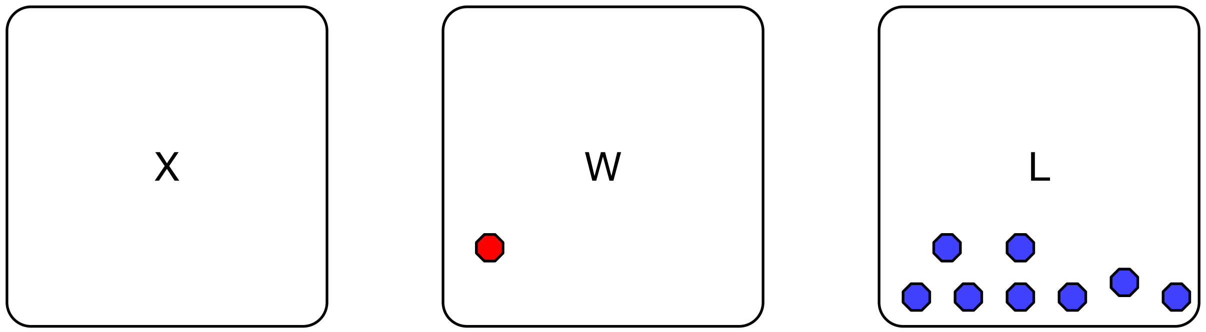
Maximum



- A single comparison can involves 6 cases
  - X-X: move two elements from X, one into W, one into L
  - X-W: move one element from X into W or move one element from X into W and one from W into L
  - X-L: move one element from X into W or one into L
  - W-W: move one element from W to L
  - W-L: nothing or move one element from W to L
  - L-L: nothing

Maximum

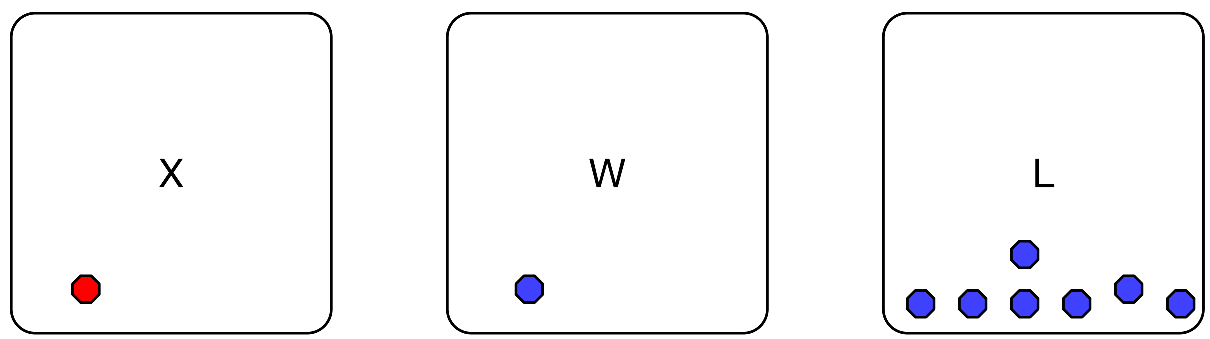
- To have finished the algorithm:
  - No elements left in X
  - Only one element left in W



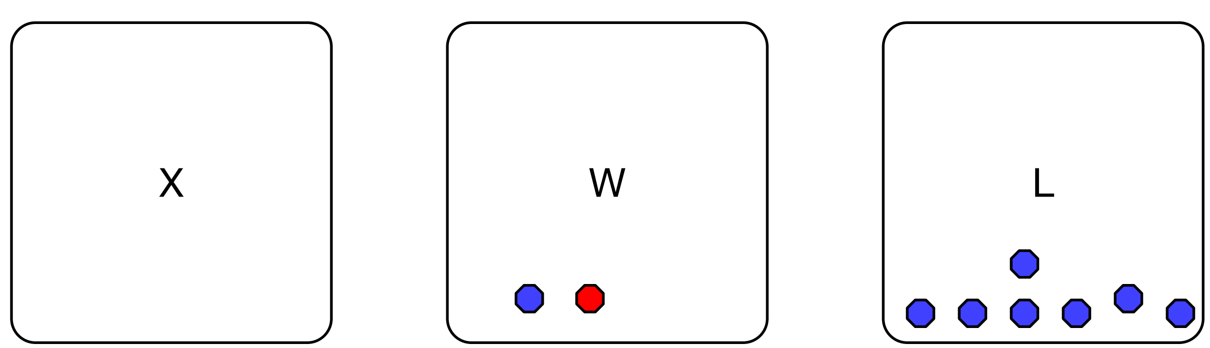
- Otherwise, can construct counterexample

Maximum

- One left in X: could be the maximum



- Two (or more) left in W:
    - Which one is the maximum?



Maximum

- Each comparison sends at most one element to L
- At best, $n - 1$ comparisons

Combined Maximum and Minimum

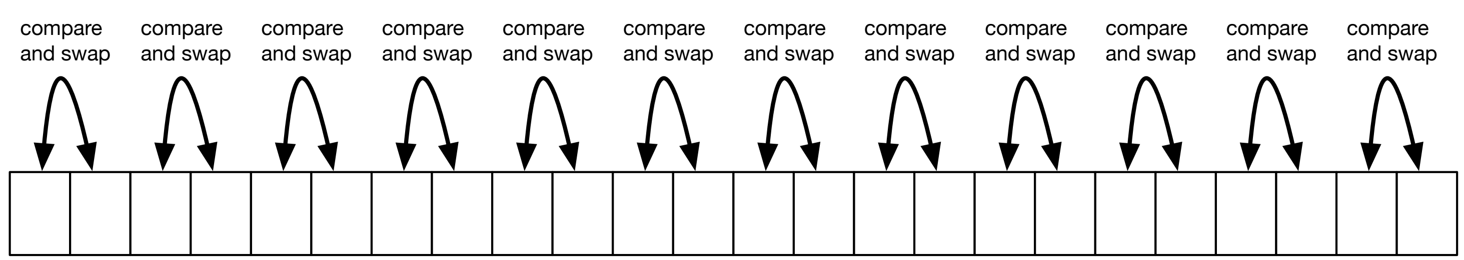
- Combined Maximum and Minimum
  - Naïve algorithm:
    - Calculate the max, then the min (can exclude the max)
      - $m - 1 + m - 2 = 2m - 3$ comparisons

Combined Maximum and Minimum

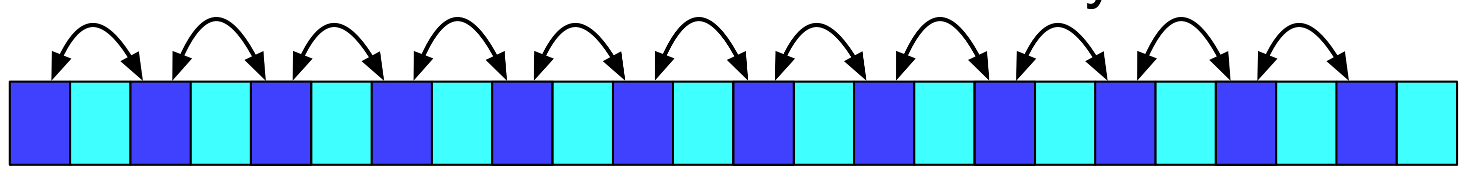
- A better algorithm
  - Divide the array into pairs
  - Compare the values of each pair
  - Place the winner of each pair in one array, the looser of each array in a second array
    - (Or use swapping so that the winners are in even position and the losers are in odd positions)
  - Now use maximum and minimum on the two sub-arrays

Combined Maximum and Minimum

- Case 1: n is even
  - There are $n/2$ pairs or $n/2$ comparisons



  - Run maximum on even indexed array elements



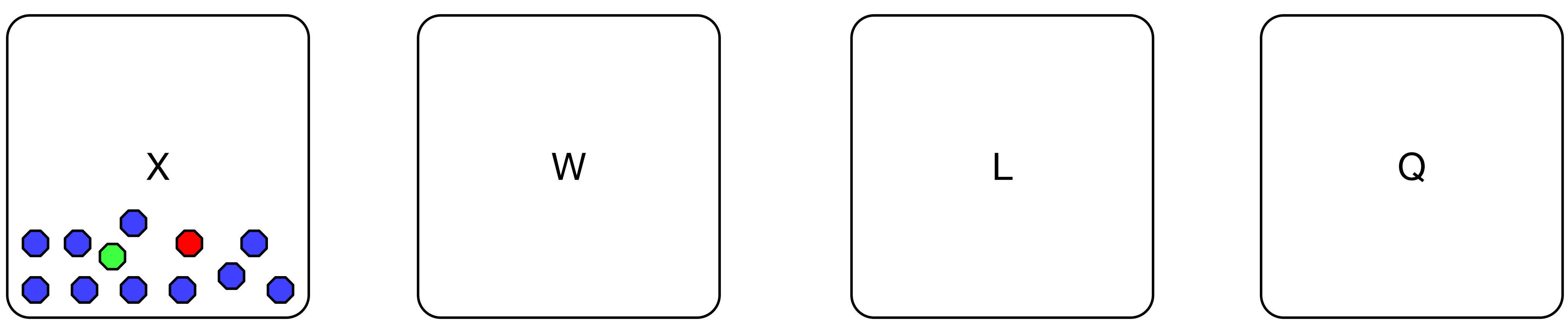
  - This gives us $n/2 - 1$ comparisons
  - Same for minimum
  - Total is $n/2 + n/2 - 1 + n/2 - 1 = \dfrac{3n}{2} - 2$ comparisons

Combined Maximum and Minimum

- Case: n is odd
  - Run algorithm on the first $n - 1$ elements
    - $\frac{3n-3}{2} - 2$ comparisons
  - Then add two comparisons to see whether the last element is either minimum or maximum
    - Total of $\frac{3n-3}{2}$ comparisons

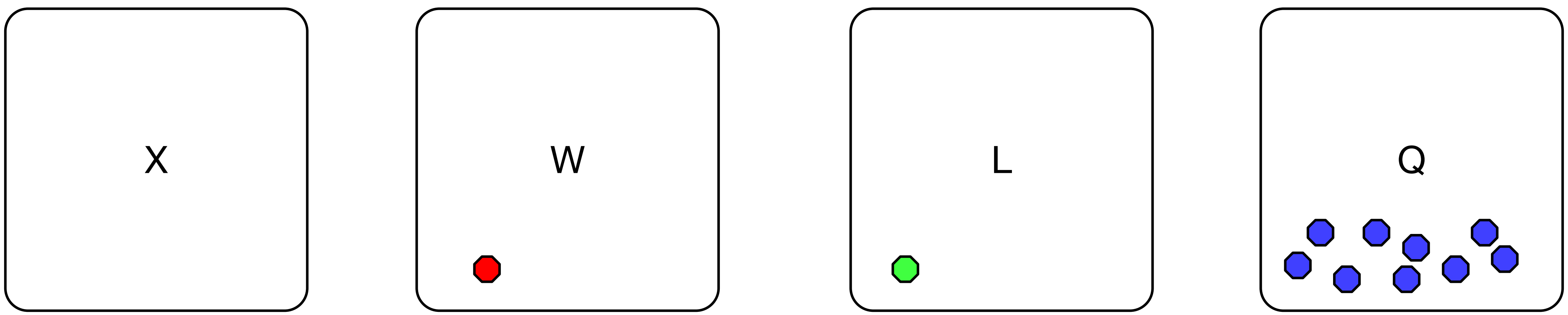
Combined Maximum and Minimum

- Can we do better?
  - Use a more sophisticated bin method
  - X - not looked at, W - won every comparison, L - lost every comparison, Q - at least one win and at least one loss



Combined Maximum and Minimum

- To be successful, need to move everything out of X and have only one element in W and L



- Otherwise can have a counter-example

Combined Maximum and Minimum

- Just counting the moves is not sufficient
  - Example:
    - We compare an element $w \in W$ with an element $l \in L$
    - Possibly: $w < l$
      - And we move both elements to the Q bucket
  - So, possible to move all n elements out of X into $W \cup L$ in $n/2$ comparisons and $n-2$ elements out of $W \cup L$ into Q in $n/2-1$ comparisons
  - Only gives $n-1$ moves!

Combined Maximum and Minimum

- Use an **adversary** argument
    - Algorithm can <u>only</u> depend on the knowledge of the <u>previous</u> comparisons when making a decision
- An adversary is allowed to change all values as long as the results of the comparisons stay the same
    - If $w \in W$ and $l \in L$, then the only thing the algorithm knows is that w has won all of its comparisons and l has lost all of its comparisons
    - Adversary therefore is allowed to change the value of l downward
    - Adversary guarantees that $w > l$.

Combined Maximum and Minimum

- With the help of the adversary who substitutes values when needed
- Potential: $\frac{3}{2}|X| + |W| + |L|$
  - Calculate net changes for comparisons between buckets

Combined Maximum and Minimum

- Compare X with X
  - Net change (-2, 1, 1, 0)
    - Potential change: 1

Combined Maximum and Minimum

- Compare X with W
  - Case 1: $x \in X, w \in W, x < w$ Net change (-1,0,1,0)
  - Case 2: $x \in X,\ w \in W,\ x > w$ Net change(-1,0,0,1)
  - The adversary can prevent Case 2 by decreasing x
    - Possible because this is the first time that we look at x
- Potential changes by $\frac{1}{2}$

Combined Maximum and Minimum

- Compare X with L
  - similar as before

Combined Maximum and Minimum

- Compare X with Q
  - The element in X changes to either W or L
    - Net change (-1, 1, 0, 0) or (-1, 0, 1, 0)
    - Potential change $\frac{1}{2}$

Combined Maximum and Minimum

- Compare W with W
  - One element looses
  - Net change (0, -1, 0, 1)
  - Potential change 1

Combined Maximum and Minimum

- Compare W with L
    - Adversary guarantees that the element in W wins by making all of them bigger
    - This works because each element in W has only seen wins and that does not change if the elements are made bigger.
    - No change

Combined Maximum and Minimum

- Compare W with Q
    - Since the elements in W have always won, the adversary can make them larger
    - No net change

Combined Maximum and Minimum

- Comparisons with L are the same as with W
- Comparisons within Q are useless, but make no changes

Combined Maximum and Minimum

- With the help of the adversary
  - Potential changes by at most 1
- Initial Potential: $\frac{3}{2}n$
- Final Potential: 2
- Need at least $\frac{3n-4}{2}$ comparisons

Selection Sort

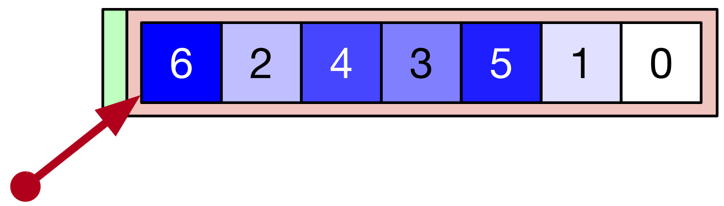
Thomas Schwarz, SJ

Selection Sort

- Divide array in sorted and unsorted parts
  - At each step, insert the minimum of the unsorted part at the end of the sorted part

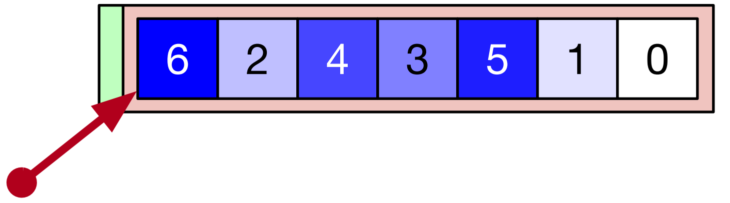
Selection Sort

- Example:



- Array is divided into a sorted (green) and unsorted portion
- We keep the index i of the first element in the unsorted portion

Selection Sort

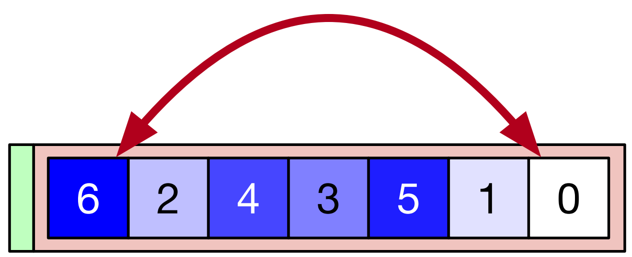
- Example:
  - 

$i = 0$
  - Find the index j of the minimum of the elements in the unsorted array
    - Implemented in numpy as argmin

```
j = np.argmin(arr[i:])+i
```

    - In Python, write your own function
  - Minimum here is 0: $j = 6$

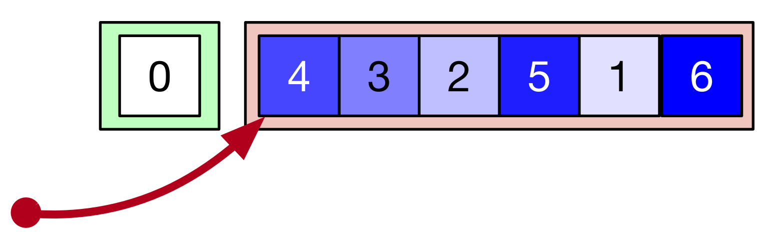
Selection Sort

- Example:
  - 

  - Now swap the elements at i and j

```
arr[i], arr[j] = arr[j], arr[i]
```

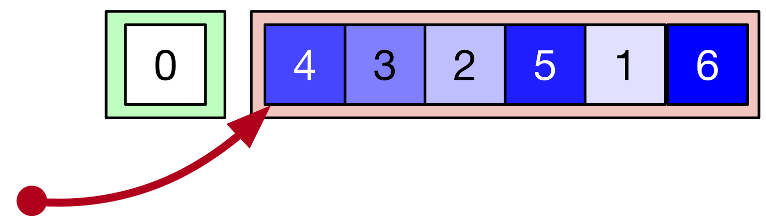
  - Increment i to 1 `i=i+1`



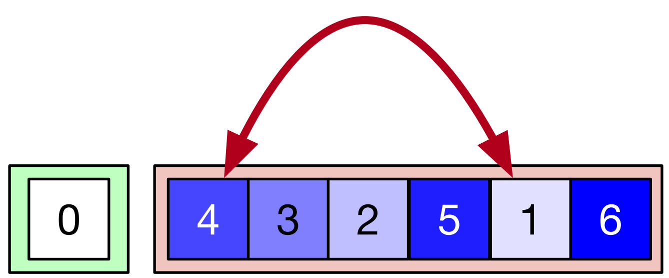
  -

Selection Sort

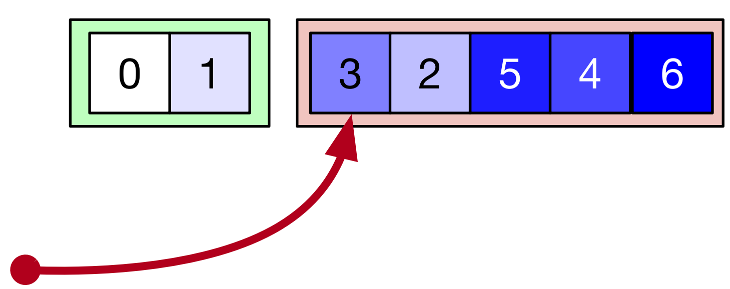
- Example:



- Minimum is now 1: $j = 5$

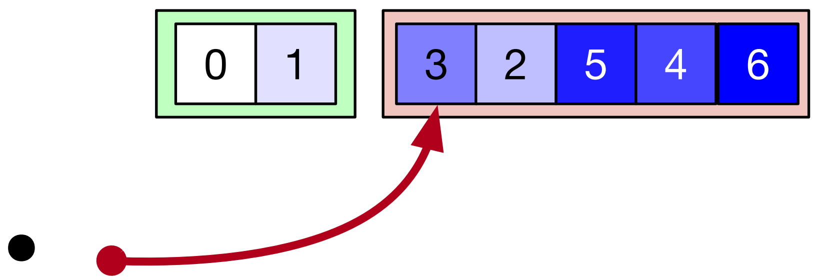


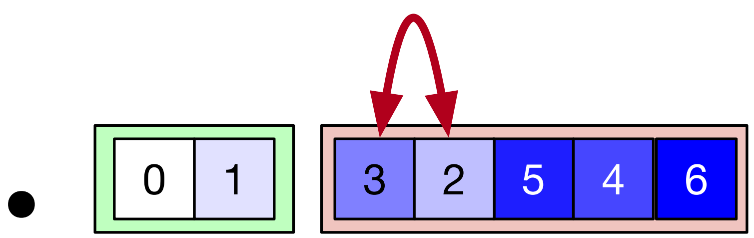
- Swap array elements at i and j and increment i

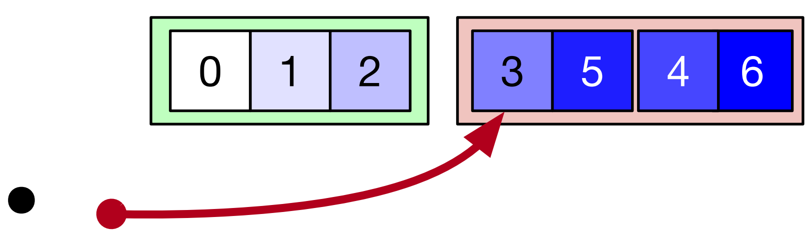


Selection Sort

- Example:

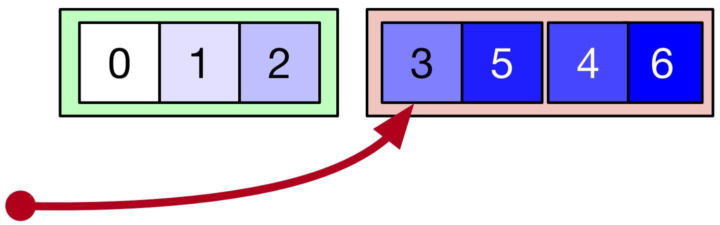


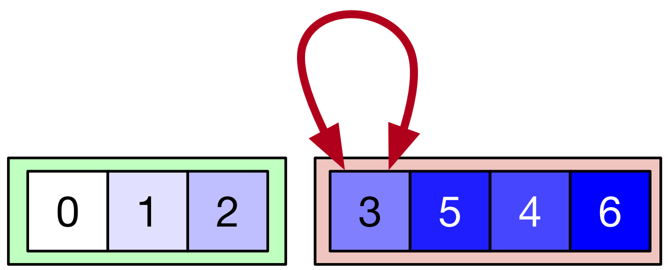


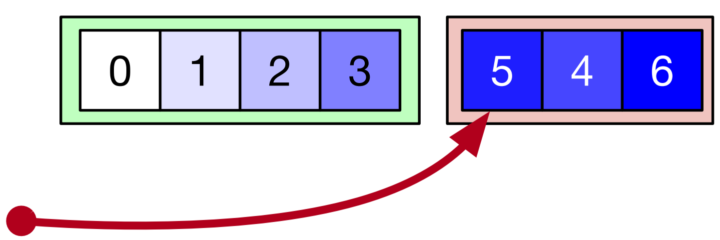


Selection Sort

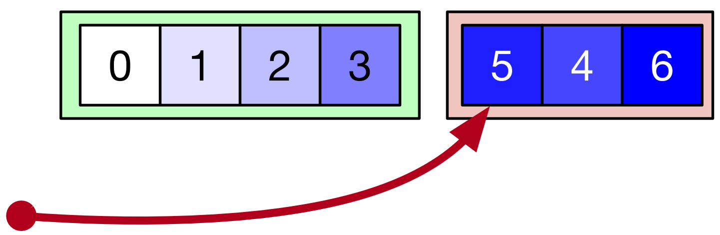
- Example:

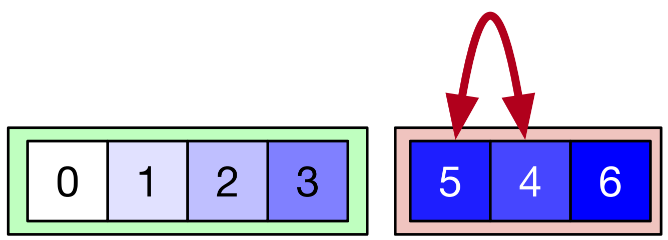


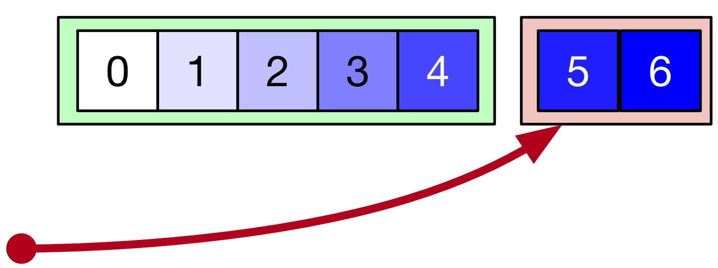




Selection Sort

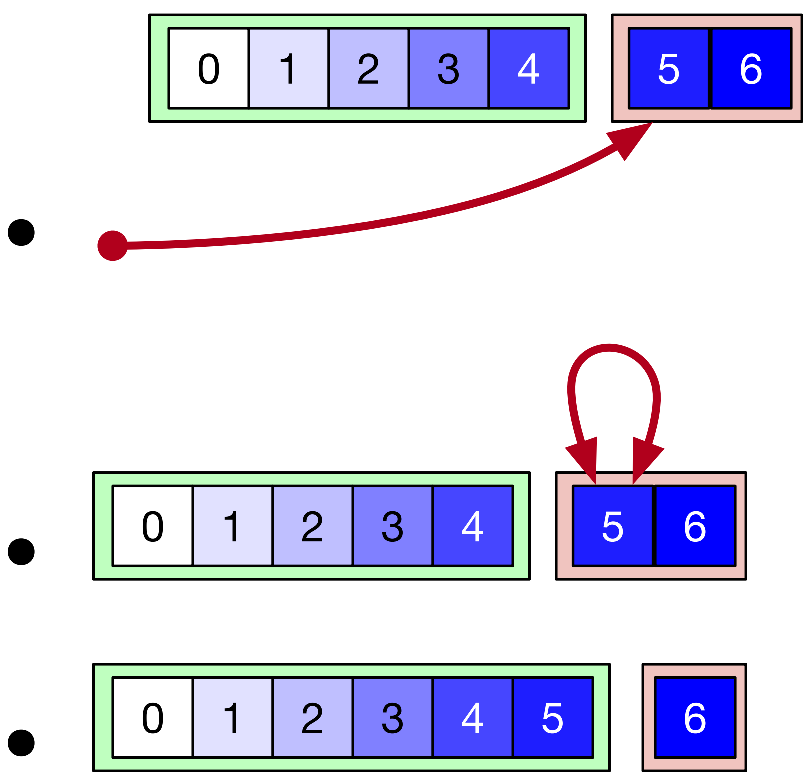
- Example:
  - 

  - 

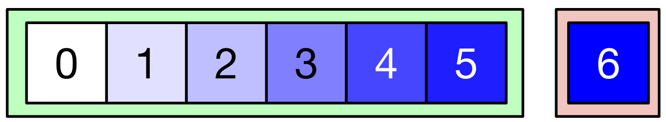
  - 

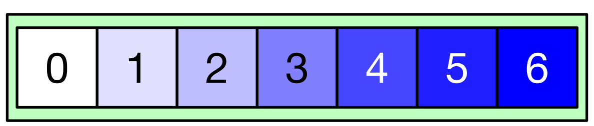
Selection Sort

- Example:



Selection Sort

- Example:
    - 

    - We can stop at $i = \text{len}(\text{arr}) - 1$ because the array is now sorted
    - 

Selection Sort

- Performance
  - At pass i:
    - Need to find minimum among $n - i$ elements
    - Costs $n - i - 1$ comparisons
  - Total costs:
    $(n-1) + (n-2) + \ldots + 2 + 1 + 0 = \frac{1}{2}n(n-1)$
    comparisons
  - $\Theta(n^2)$

Selection Sort

- In practice:
  - Among quadratic sorting algorithms:
    - Usually the best performing one

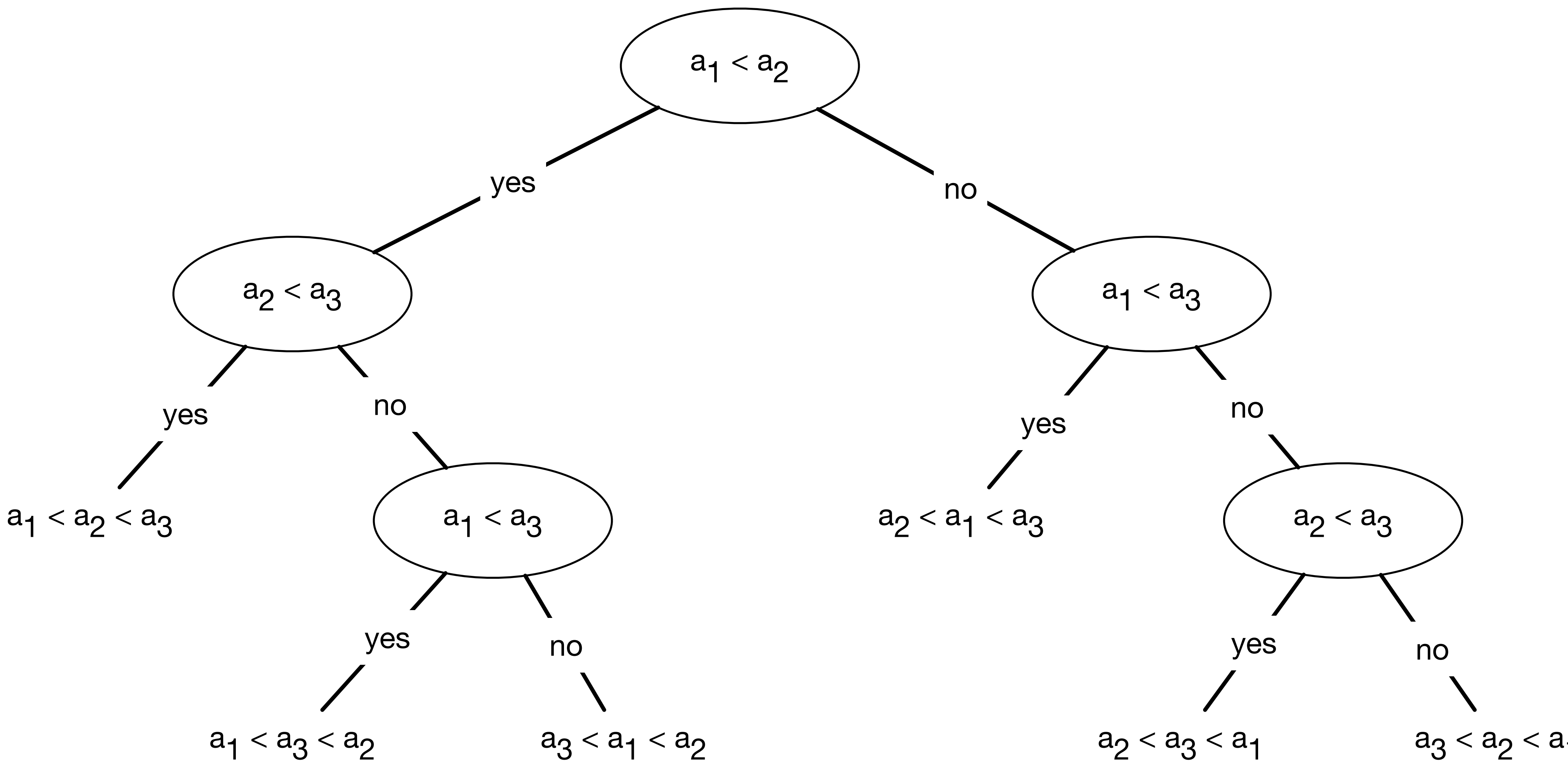
Sorting by Comparison

- Many sorting algorithms use comparisons
- An algorithm needs to be able to sort with all orders of inputs, i.e. distinguish between $n!$ arrangements of the input by order
    - assuming all elements are different

Sorting by Comparison

- Sorting algorithm makes a comparison, then decides on what to do
- Can be represented as a binary tree

Sorting by Comparison



A fictitious algorithm for sorting three elements as a Decision Tree

Sorting by Comparison

- Represent any comparison based algorithm by such a tree
- Any run of the algorithm represents a path from the root to a leaf node
- Leaf nodes represent an algorithm finishing,
    - So they need to have an ordering, i.e. a permutation of the input array

Sorting by Comparison

- How many steps does a tree with N leaves have?
- A tree of height h has how many leaves?
  - Height 0: only root, one leaf
  - Height 1: only root plus one or two leaves: ≤ 2
  - Height 2: at most two nodes at height one have at most $\leq 2^2$ leaves
  - Induction: Height h has at most 2^h leaves

Sorting by Comparison

- Relationship between height of decision tree and number of elements to be sorted:
  - Need to have at least $n!$ leaves:
  - $2^h \geq n!$
    - which implies
      - $h \geq \log_2(n!) = \dfrac{1}{\log(2)} \log(n!)$
      - $\approx \dfrac{1}{\log(2)} n \log(n) - n + 1$
      - $= \Theta(n \log(n))$

Sorting by Comparison

- Since the height of the decision tree is the worst time runtime, we have

  - *The runtime of a comparison based sorting algorithm is* **at least** $\Theta(n \log(n))$

Better Sorting Algorithms

Heap-Sort

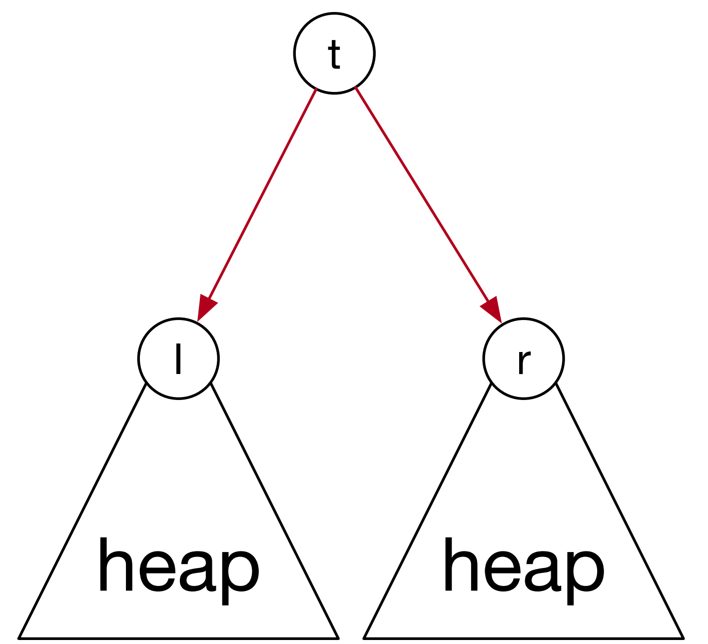
- Example of a sorting algorithm that uses additional space
- But variants make it an in-place algorithm
  - A version of selection sort with the right data structure for the unordered part
  - Idea:
    - Insert all elements into a heap
    - Then empty the heap with calls of extract-min
    - Get the same elements back, but in order
  - Performance:
    - $\approx \log(1) + \log(2) + \ldots + \log(n-1) + \log(n) \approx \log(n)n$

Heap-Sort

- Details:
  - Step 1: convert array into a maximum heap
    - Idea:
      - Elements in the second half are all leaves
      - Form their own sub-heaps
      - Need to learn how to convert two sub-heaps and a parent into a proper head

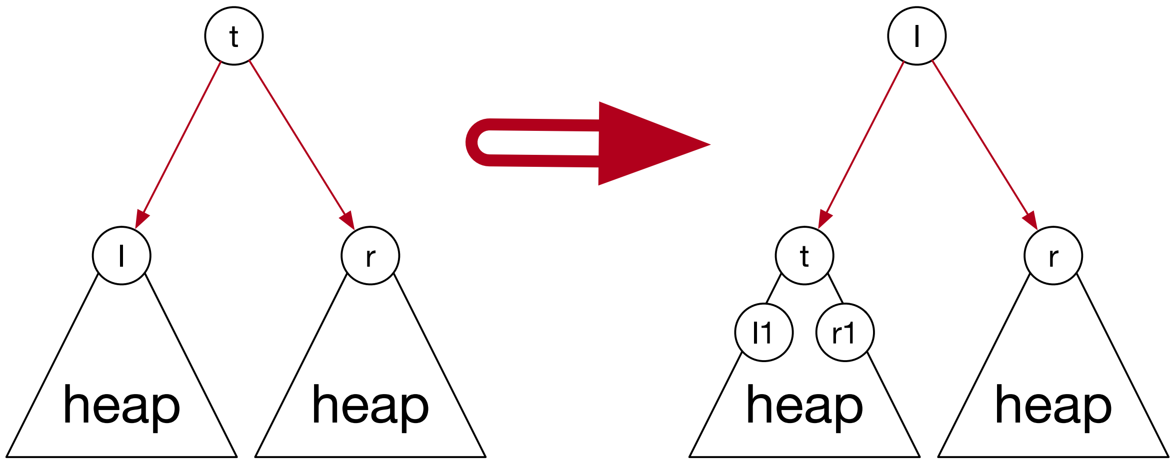
Heap-Sort

- How to heapify two sub-heaps?



- If $t \geq l$ and $t \geq r$: ensemble already a heap
- If $l = \max(\{t, l, r\})$: exchange t and l
    - But now the left might no longer be a heap

Heap-Sort



- Because the root of the left heap has become smaller, the heap property there is no longer guaranteed
- We need to continue heapifying there

Heap-Sort

```
def heapify(arr, i):
   l, r = left(i), right(i)
   if l < len(arr):
      if arr[i] < arr[l]:
         largest = l
      else:
         largest = i
   if r < len(arr):
      if arr[r] > arr[largest]:
         largest = r
   if largest == i:
      return
   else:
      arr[i], arr[largest] = arr[largest], arr[i]
      heapify(arr, largest)
```

Heap-Sort

- Performance of heapify: $O(\log(n))$
- To guarantee result is a heap:
    - left and right subheap need to be heaps indeed

Heap-Sort

- To create a heap:
  - use heapify working back
  - Can start at location $\lfloor \text{len(arr)}/2 \rfloor$

```
def make_heap(arr):
   for i in range( int(len(arr)/2), 0, -1):
      heapify(arr,i)
```

- We can even show: runtime of make_heap is linear

Heap-Sort

- Heap-sort:
  - Make array into a heap
  - Extract the maximum
    - move it to the last element of the array
  - Repeat

Heap-Sort

```
def heap_sort(arr):
    for i in range(len(arr)-1, 1, -1):
        arr[0], arr[i] = arr[i], arr[0]
        arr.heap_size = arr.heap_size - 1
        heapify(arr, i)
```

- Performance:
    - Making the array into a heap: $O(n)$
    - Extracting the maximum and putting it at the end: $\Theta(1)$
    - Heapify the array again:
      $O(\log_2(n) + O(\log_2(n-1)) + \ldots + O(\log_2(2)) + O(\log_2(1)))$

Heap Sort Example

- Example
  -

9	8	3	4	11	6	13	1	7	10	12	0	2	5

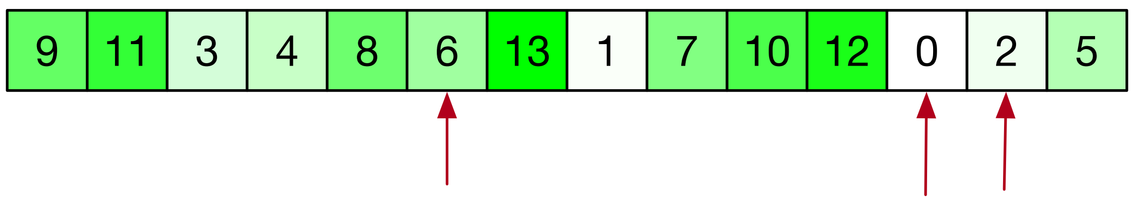
  - First phase: heapify into a max heap
    - Easier to start indices with 1
    - $j = 7, \, l = 14, \, r = 15$
    -

9	11	3	4	8	6	13	1	7	10	12	0	2	5

    - Heap property is true

Heap Sort Example

- $j = 6,\ l = 12,\ r = 13$



-

- Heap property maintained

Heap Sort Example

- $i = 5$, left $= 10$,right $= 11$

9	11	3	4	8	6	13	1	7	10	12	0	2	5

- Heap property needs to be restored:
  - Exchange 8 for 12

9	11	3	4	12	6	13	1	7	10	8	0	2	5

  - No need to continue

Heap Sort Example

- $j = 4,\ l = 8,\ r = 9$
  -

9	11	3	4	12	6	13	1	7	10	8	0	2	5

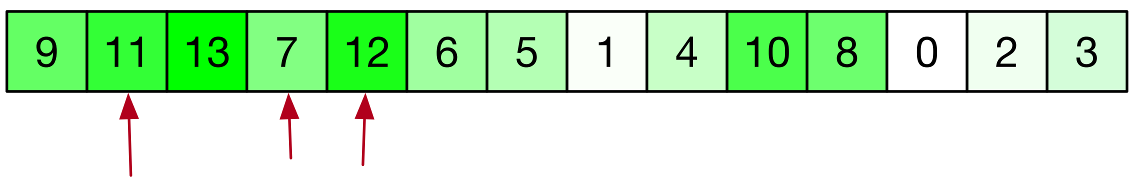
  - Exchange 4 with 7
  -

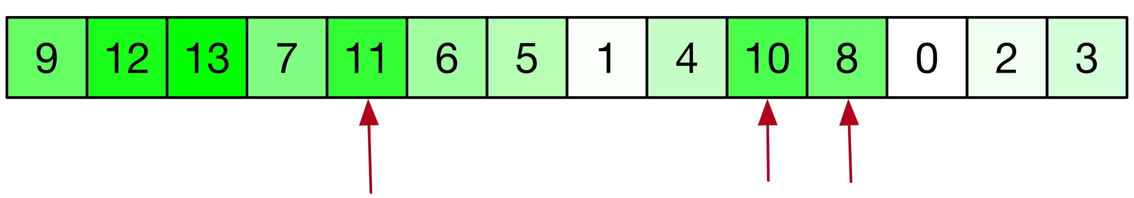
9	11	3	7	12	6	13	1	4	10	8	0	2	5

Heap Sort Example

- $j = 3,\ l = 6,\ r = 7$
  -
9	11	3	7	12	6	13	1	4	10	8	0	2	5
  - Exchange 3 with 13
  -
9	11	13	7	12	6	3	1	4	10	8	0	2	5
  - Test result $j = 7,\ l = 14$: exchange 3 with 5
  -
9	11	13	7	12	6	5	1	4	10	8	0	2	3

Heap Sort Example

- $j = 2,\ l = 4,\ r = 5$
- 

- Exchange 11 with 12
- Then check heap property with $i = 4,\ l = 8,\ r = 9$
- 

Heap Sort Example

- $j = 1, \ l = 2, \ r = 3$

9	12	13	7	11	6	5	1	4	10	8	0	2	3

- Exchange 9 with 13 and check $i = 3, \ l = 6, \ r = 7$

13	12	9	7	11	6	5	1	4	10	8	0	2	3

Heap Sort Example

- Second phase:
  - Extract maxima:
13	12	9	7	11	6	5	1	4	10	8	0	2	3
  - Exchange 13 with 3 and heapify:
3	12	9	7	11	6	5	1	4	10	8	0	2	13
    - Exchange 3 with 9
12	3	9	7	11	6	5	1	4	10	8	0	2	13
    - Exchange 3 with 11
12	11	9	7	3	6	5	1	4	10	8	0	2	13

Heap Sort Example

12	11	9	7	3	6	5	1	4	10	8	0	2	13
- Exchange 3 with 10
12	11	9	7	10	6	5	1	4	3	8	0	2	13
- Can stop here

Heap Sort Example

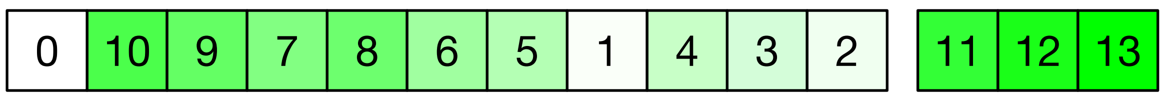
- Extract maximum again
  - Exchange 12 with last element of heap
2	11	9	7	10	6	5	1	4	3	8	0	12	13
  - Now heapify again
11	2	9	7	10	6	5	1	4	3	8	0	12	13

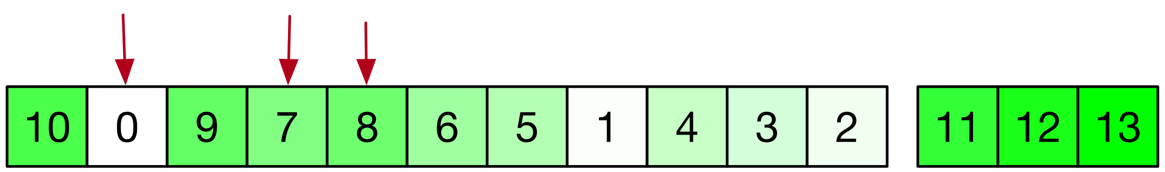
    (change 2 with 10)
11	10	9	7	2	6	5	1	4	3	8	0	12	13

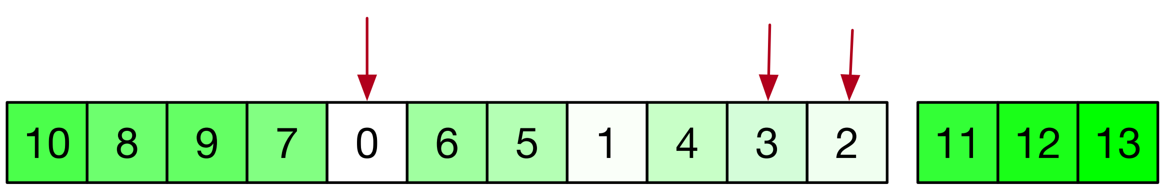
    (change 2 with 8)
11	10	9	7	8	6	5	1	4	3	2	0	12	13

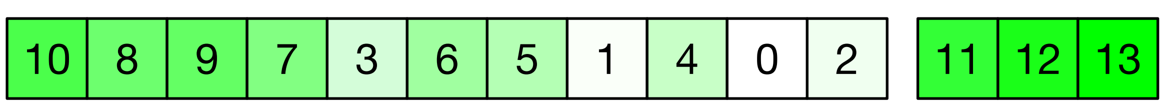
    (can stop here)

Heap Sort Example

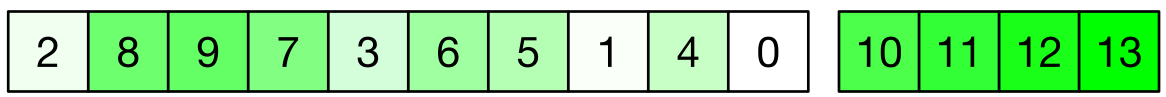
- Extract maximum:
  - 

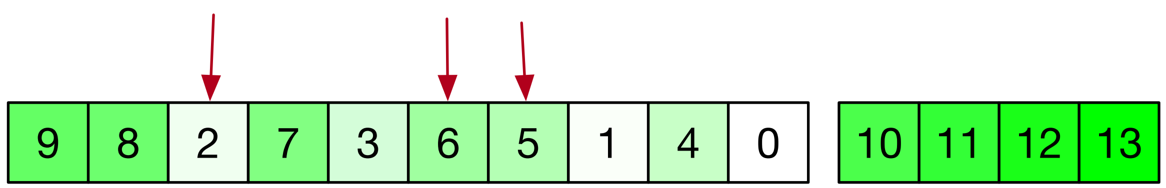
  - Heapify:
  - 

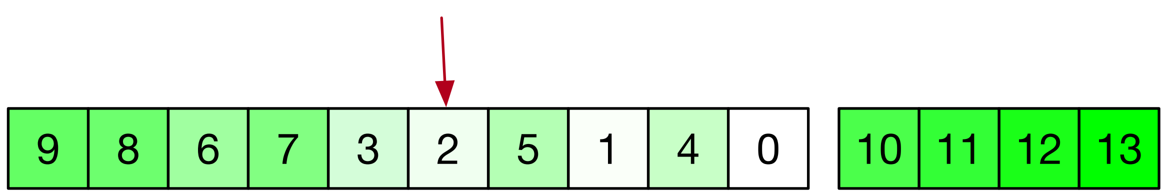
  - 

  - 

Heap Sort Example

- Extract maximum
  - 

- Heapify
  - 

  - 

  - We can stop here because the left and right index point to elements outside the heap

Heap Sort Example

- Extract maximum
  -

0	8	6	7	3	2	5	1	4	9	10	11	12	13

- Heapify
  -

8	0	6	7	3	2	5	1	4	9	10	11	12	13

  -

8	7	6	0	3	2	5	1	4	9	10	11	12	13

  -

8	7	6	4	3	2	5	1	0	9	10	11	12	13

Heap Sort Example

- Extract maximum
  - | 0 | 7 | 6 | 4 | 3 | 2 | 5 | 1 | | 8 | 9 | 10 | 11 | 12 | 13 |
- Heapify
  - | 7 | 0 | 6 | 4 | 3 | 2 | 5 | 1 | 8 | | 9 | 10 | 11 | 12 | 13 |
  - | 7 | 4 | 6 | 0 | 3 | 2 | 5 | 1 | | 8 | 9 | 10 | 11 | 12 | 13 |
  - | 7 | 4 | 6 | 1 | 3 | 2 | 5 | 0 | | 8 | 9 | 10 | 11 | 12 | 13 |

Heap Sort Example

- Extract maximum
  -

0	4	6	1	3	2	5	7	8	9	10	11	12	13

- Heapify
  -

6	4	0	1	3	2	5	7	8	9	10	11	12	13

  -

6	4	5	1	3	2	0	7	8	9	10	11	12	13

Heap Sort Example

- Extract maximum
  -

0	4	5	1	3	2	6	7	8	9	10	11	12	13

- Heapify
  -

5	4	0	1	3	2	6	7	8	9	10	11	12	13

  -

5	4	2	1	3	0	6	7	8	9	10	11	12	13

Heap Sort Example

- Extract maximum
  -
0	4	2	1	3	5	6	7	8	9	10	11	12	13
- Heapify
  -
4	0	2	1	3	5	6	7	8	9	10	11	12	13
  -
4	3	2	1	0	5	6	7	8	9	10	11	12	13

Heap Sort Example

- Extract maximum

0	3	2	1		4	5	6	7	8	9	10	11	12	13

- Heapify

3	0	2	1		4	5	6	7	8	9	10	11	12	13

3	1	2	0		4	5	6	7	8	9	10	11	12	13

Heap Sort Example

- Extract maximum
  -

0	1	2		3	4	5	6	7	8	9	10	11	12	13

- Heapify
  -

2	1	0		3	4	5	6	7	8	9	10	11	12	13

Heap Sort Example

- Extract maximum
  -

0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Heapify
  -

1	0	2	3	4	5	6	7	8	9	10	11	12	13

Heap Sort Example

- Extract maximum
0		1	2	3	4	5	6	7	8	9	10	11	12	13
- Extract maximum
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Linear Time Sorting

- Counting sort
  - Assume we want to sort numbers in $\{1,2,\ldots,k-1,k\}$
  - Create a dictionary with keys in $\{1,2,\ldots,k-1,k\}$
    - E.g. as an array `Int(1:k)`
  - Walk through the array, updating the count
  - Once the count is done, go through the dictionary in order of the keys, emitting as many keys as the count

Linear Time Sorting

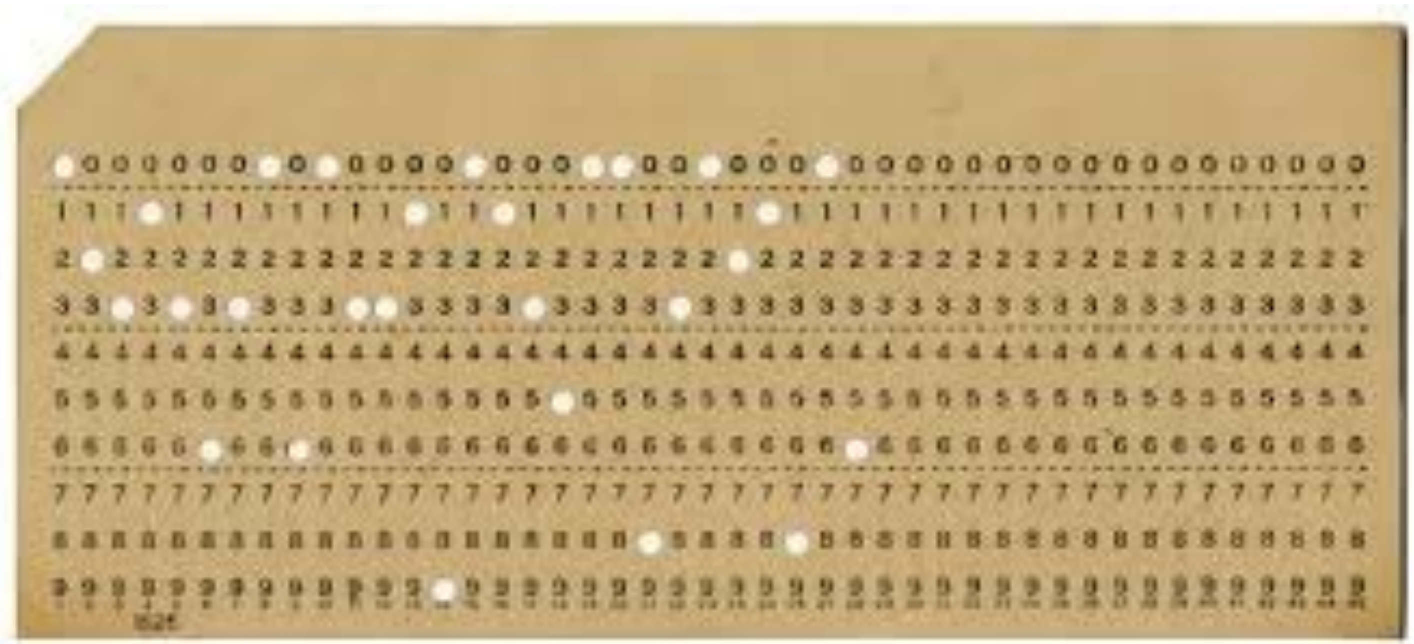
- Counting sort:
10	3	4	10	12	4	5	3	8	9	2	2	5	10	1	2	7
  - create a counting array:
1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:	12:	13:
  - Walk through the array and calculate counts
1: 1	2: 3	3: 3	4: 2	5: 2	6: 0	7: 1	8: 1	9: 1	10: 3	11: 0	12: 1	13: 0
  - Emit keys according to count
    - 1 2 2 2 3 3 3 4 4 5 5 7 8 9 10 10 10 12

Linear Time Sorting

- If there are n elements in the array, then counting sort uses
  - $\sim k$ to create and evaluate the counting array
  - $\sim n$ to update the counting array
- Therefore: counting sort run-time is $\Theta(n + k)$

Linear Time Sorting

- Radix Sort
  - Imagine sorting punch cards with by ID in the first columns



Linear Time Sorting

- Simple Method:
  - Create heaps of cards based on the first digit
    - Then recursively sort the heaps

Linear Time Sorting

- Better method:
  - Sort according to the last digit
    - Then use a *stable sort* to sort after the second-last digit
    - Then use a stable sort to sort after the third-last digit

Linear Time Sorting

- Stable sort:
  - Leave order of elements with the same key during sorting
  - Insertion sort, merge sort, bubble sort, counting sort are all stable
  - Heap sort, selection sort, shell sort, and quick sort are not

Linear Time Sorting

- Radix sort:
    - ```
      for i in range(length(key), 0, -1):
          stable_sort on digit i of key
      ```
      ```

Linear Time Sorting

135	220	302	023
242	321	203	122
122	221	220	135
023	242	321	144
220	122	221	203
144	302	122	220
321	023	023	221
221	203	135	242
203	144	242	302
302	135	144	321

Linear Time Sorting

- Radix sort correctness
  - What would be a loop invariant?

Linear Time Sorting

- Assume n keys of d digits in $\{0,1,\ldots,r-1\}$
- Use counting sort to sort in time $\Theta(n+r)$
- Radix sort then takes $\Theta(d(n+r))$ time

Linear Time Sorting

- Given n numbers of b bits each
- Assume $b = O(\log(n))$
- Choose $r = \lfloor \log_2(n) \rfloor$.
  - Divide the b-bit numbers into "digits" of length r
  - Thus, each round of radix sort takes time $\Theta(n + 2^r)$
  - There are $\lceil \frac{b}{r} \rceil$ rounds
  - So, radix sort takes $\Theta(\frac{b}{r}(n + 2^r)) = \Theta(\frac{b}{r}(n + n)) = \Theta(n)$ time!