Analysis of Euclidean Algorithm

Algorithms

Thomas Schwarz, SJ

Greatest Common Divisor

- Given two numbers $a, b \in \mathbb{N}$:
  - a divides b $\quad a \mid b :\Longleftrightarrow \exists x \in \mathbb{N}: \; b = ax$
    - Divisors are smaller than the dividend
      - $a \mid b \implies a \leq b$
  - r is a common divisor of a and b iff $r \mid a \wedge r \mid b$
  - $\text{gcd}(a, b) = \max\{r : r \mid a \wedge r \mid b\}$
    - Always exists because the set is finite
    - Any finite subset of the natural numbers has a maximum

Greatest Common Divisor

- Lemma 1: For all numbers $a, b \in \mathbb{N}$:
  gcd$(a, b) =$ gcd(b, a)
- Proof: The set of common divisors does not depend on the order in which a and b are given:
  - $\{r : r \mid a \wedge r \mid b\} = \{r : r \mid b \wedge r \mid a\}$ because the logical and operator is commutative

    Hence: gcd$(a, b) = \max\{r : r \mid a \wedge r \mid b\}$

    $= \max\{r : r \mid b \wedge r \mid a\}$

    $=$ gcd(b, a)

Greatest Common Divisor

- Lemma 2: If $a \in \mathbb{N}$ and $a \mid b$ then gcd$(a, b) = a$.
- Proof:
  - a is the largest divisor of itself.
  - a is also a divisor of b by assumption
  - Hence a is the largest element in the set of common divisors $\{r : r \mid a \wedge r \mid b\}$.
  - This means that $a = \max\{r : r \mid a \wedge r \mid b\} =$ gcd(a, b)

Greatest Common Divisor

- Lemma 3: If $a \equiv c \pmod{b}$ then $\text{gcd}(a, b) = \text{gcd}(c, b)$
- Proof:
    - $a \equiv c \pmod{b} \iff \exists r, s, t \in \mathbb{N}_0 : a = rb + t \wedge c = sb + t \wedge 0 \leq t < b$
    - We show that $\{r : r \mid a \wedge r \mid b\} = \{r : r \mid c \wedge r \mid b\}$
    - Assume that $d \in \mathbb{N}$ is in the left side. We want to show that it is also in the right side. For this we need to show that d also divides c.
        - What do we know: There exists $x, y \in \mathbb{N}_0$ such that
            - $b = xd$ because d divides b
            - $a = yd$ because d divides a
            - $a = rb + t, c = sb + t, 0 \leq t < b$

Greatest Common Divisor

- Proof (continued)
  - $$\begin{aligned} c &= c - a + a \\ &= ((sb + t) - (rb + t)) + a \\ &= (s - r)b + a \\ &= (s - r)xd + yd \\ &= ((s - r)x + y)d \end{aligned}$$

Greatest Common Divisor

- Proof: (cont)
  - Now we want to show that all elements on the right side of $\{r : r \mid a \wedge r \mid b\} = \{r : r \mid c \wedge r \mid b\}$ are in the left side.
  - However, since our assumptions are symmetric in a and c, the same proof applies.

Euclidean Algorithm

- Informal Version:
  - To compute gcd(a, b) put the larger number of a and b on the left
  - Then divide a by b with remainder r ($a = bx + r$)
    - If $r = 0$, then $b \mid a$ and gcd$(a, b) = b$.
  - Otherwise:
    - Notice that $r \equiv a \pmod{b}$.
    - Therefore gcd$(a, b) =$ gcd$(r, b) =$ gcd(b, r) by the Lemma
  - Continue until the remainder becomes 0

Euclidean Algorithm

- gcd(1043, 4321)
    - = gcd(4321, 1043)
    - = gcd(1043, 149)
    - = 149 because 1043 % 149 = 0.
- There is an interesting extension:
    - 4321=4*1043+149, ergo 149 = 4321-4*1043, a linear combination of 4321 and 1043

Euclidean Algorithm

gcd(198, 168)

= gcd(168, 30)

= gcd(30, 18)

= gcd(18,12)

= gcd(12,6)

= 6

- 198-168=30
- 18 =168 - 5*30 =168-5(198-168)=6*168-5*198
- 12 = 30 - 18 = 198-168-6*168+5*198 = 6*198-7*168
- 6 = 18-12 = -5*198+6*168-6*198+7*168 = -11*198 + 13*168
- GCD is a linear combination of the two parameters!

Euclidean Algorithm

- Pseudo-code

```
def gcd(a, b):
    if b==0:
        return a
    else:
        return gcd(b, a%b)
```

Euclidean Algorithm

- How do we prove the correctness of an algorithm?
  - Especially if it contains a loop
    - Usually, need to use induction
      - Sometimes using a *loop invariant*

```
gcd(198, 168)
= gcd(168, 30)
= gcd(30, 18)
= gcd(18,12)
= gcd(12,6)
= gcd(6,0)
```

- In this case: gcd(var1,var2) does not change between between calls
- That is Lemma 3!

- End if the algorithm ever ends, it prints out the correct value by Lemma 1.

Euclidean Algorithm

- How do we prove the correctness of the algorithm?
  - It is possible that an algorithm will never stop
    - (on some inputs, or on all inputs)
  - In our case, the smaller of the variables becomes strictly smaller
    - with the exception of the first step
  - Thus, we will run out of variables for our recursive calls sooner or later
- Algorithm will eventually return the correct number

Euclidean Algorithm

- Performance
  - Obviously, proportional to the number of recursive calls
  - Given two random inputs:
    - Can stop in one iteration
      - If second variable divides the first
    - Or can stop after many
  - In a case like this: look for the worst case scenario

Euclidean Algorithm

- Theorem: If gcd(a,b) makes *N* recursive calls and a > b then $a \geq f_{N+2}$ and $b \geq f_{N+1}$

Euclidean Algorithm

```
def gcd(a, b):
    if b==0:
        return a
    else:
        return gcd(b, a%b)
```

- Proof:
  - By induction
    - **Base case:** $N = 1$:
      - In this case $b \neq 0$, hence $b \geq 1 = f_1$
      - In this case $a > b$, so $a > b = 1 \implies a \geq 2 = f_2$

Euclidean Algorithm

```
def gcd(a, b):
    if b==0:
        return a
    else:
        return gcd(b, a%b)
```

- Induction step
  - Induction hypothesis:
    - If gcd has N recursive calls then $a \geq f_{N+2}$ and $b \geq f_{N+1}$
  - To show:
    - If gcd has $N + 1$ recursive calls, then $a \geq f_{N+3}$ and $b \geq f_{N+2}$

Euclidean Algorithm

```
def gcd(a, b):
    if b==0:
        return a
    else:
        return gcd(b, a%b)
```

- Assume that gcd(a,b) makes N+1 calls.
- The first step calls gcd(b,a%b)
  - This call calls the function recursively N times
    - Thus, by Induction Hypothesis
      - $b \geq f_{N+2}$ and $a \,\%\, b \geq f_{N+1}$
      - By division with reminder $a = rb + a \,\%\, b$ with $0 \leq a \,\%\, b < b$
        - Because $a > b$ we have $r \geq 1$.
      - Therefore: $a \geq b + a \,\%\, b \geq f_{N+2} + f_{N+1} \geq f_{N+3}$.
      - We already know that $b \geq f_{N+2}$

Euclidean Algorithm

```
def gcd(a, b):
    if b==0:
        return a
    else:
        return gcd(b, a%b)
```

- Can find a closed form of Fibonacci
  - $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.68$
  - $b \geq f_{N+2} \geq \Phi^N$
  - This implies that $\log_{\Phi}(b) \geq N - 1$ and $N = O(\log b)$

Loop Invariants

- Recursion usually demands induction proofs to assert properties of an algorithm
- For loops, use loop invariant:
  - A property that is true before the loop starts
  - A property that remains true after each loop iteration
  - And is therefore true after the loop terminates

Loop Invariants

- Working with loop invariants:
  - Need to come up with a loop invariant
  - Prove that it is true before the loop starts (induction base)
  - Prove that it remains true after each iteration of the loop

Loop Invariants

- Trivial Example:
  - Small C-program

```
extern int c;
int x = c, y = 0;
while (x>=0):
    x--;
    y++;
print(y)
```

Loop Invariants

- Step 1: Guessing a loop invariant

```
extern int c;
int x = c, y = 0;
while (x>=0):
    x--;
    y++;
print(y)
```

- Needs to involve x, y, c
  - $x + y = c$

Loop Invariants

- Step 2:
  - Show that it is true before the loop starts
    - Simple: before the loop starts, we have $x = c, y = 0$ therefore $x + y = c$

Loop Invariants

- Step 3: Show that the truth does not change after one iteration
  - Induction step: Assume $x_b + y_b = c$ before the loop iteration
    - After the iteration, we have $x_a = x_b - 1, y_a = y_b + 1$.
    - This implies
    - $x_a + y_a = (x_b - 1) + (y_b + 1) = x_b + y_b - 1 + 1 = x_b + y_b = c$

Loop Invariants

- Step 4: Evaluate with the loop invariant
  - When the loop is terminated, $x = 0$.
    - (Question: why do we now that the loop terminates?)
    - Therefore, the value of y is $y = x + y - x = c - 0 = c$
    - Thus, the function prints out the value of c.

Examples

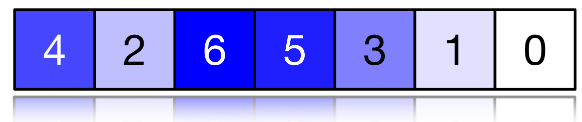
Bubble Sort

Thomas Schwarz, SJ

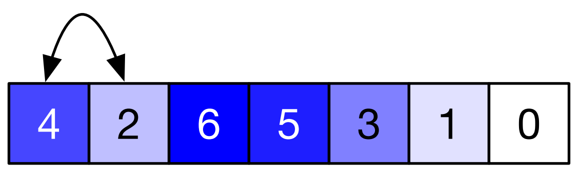
Bubble Sort

- 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]
```

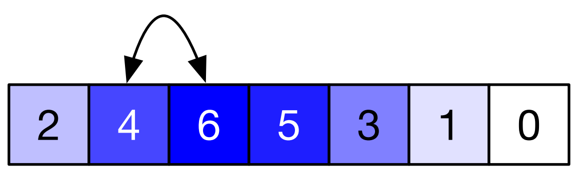
Bubble Sort

- Example: Sort 

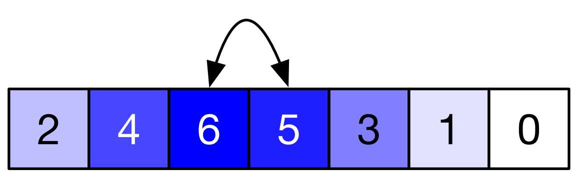
    - First pass: Check first pair



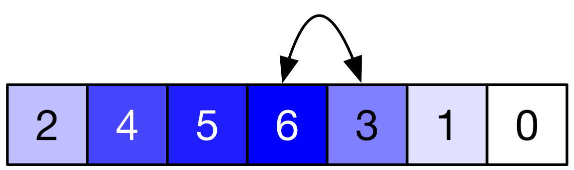
        - Swap and move on

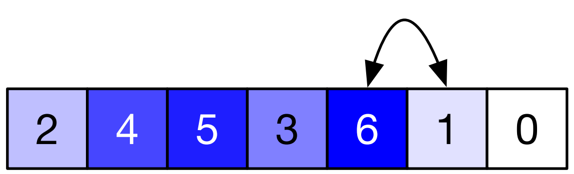


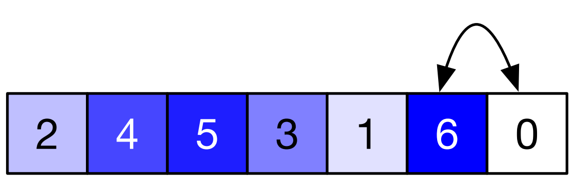
        - No swap necessary, move on



Bubble Sort

- Example:
  - Swap and move on
    - 

  - Swap and move on
    - 

  - Swap and move on
    - 

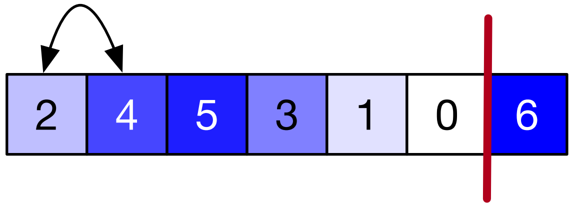
Bubble Sort

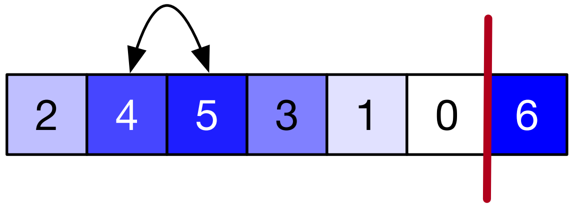
- Example:
  - Swap and move on
    -

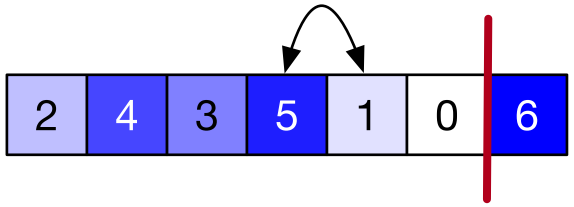
2	4	5	3	1	0	6

  - 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

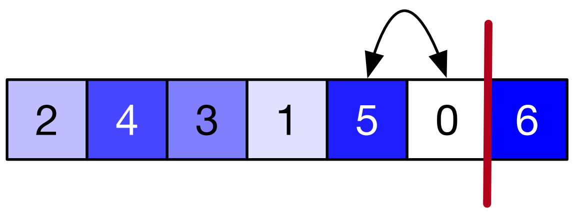
Bubble Sort

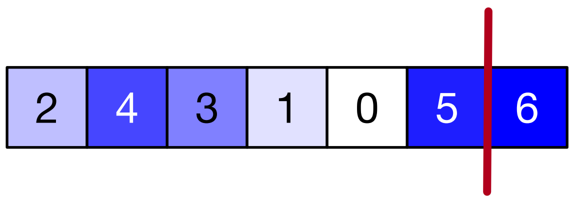
- Example
  - Second pass:
    - 

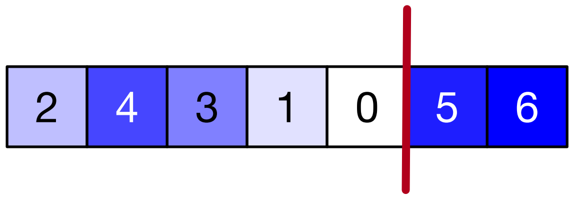
    - 

    - 

Bubble Sort

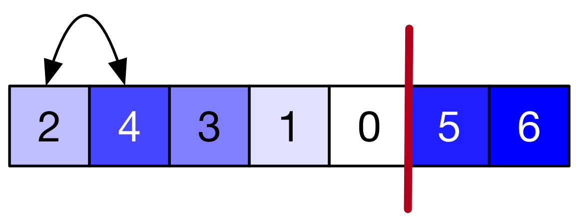
- Example (Second Pass):
  - 

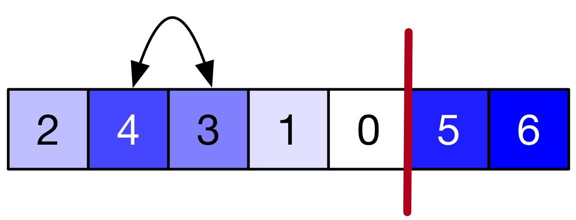
  - 

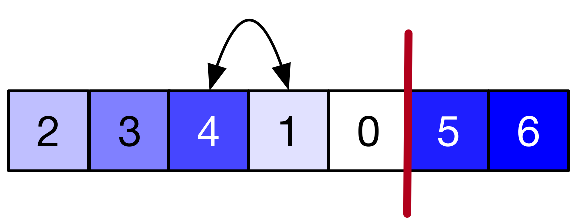
- The maximum in the remaining array has now reached its correct point
  - 

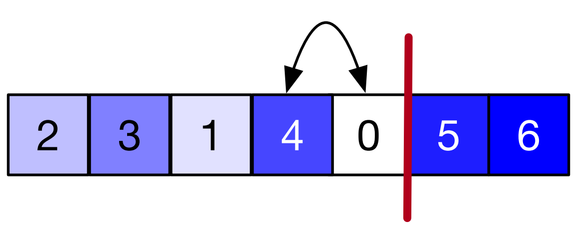
Bubble Sort

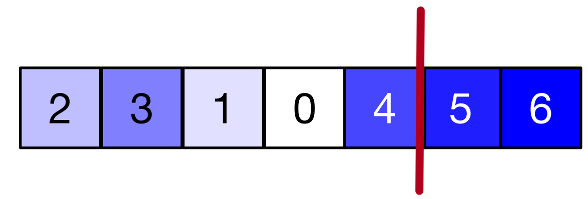
- Example: Third Pass

- 

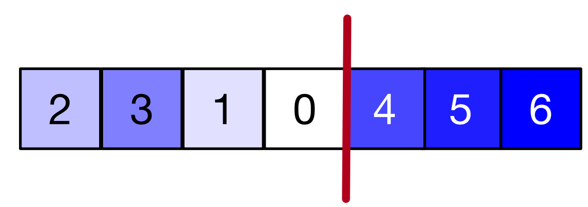
- 

- 

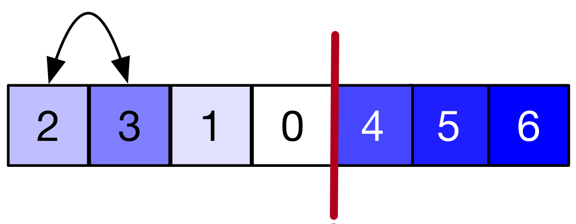
- 

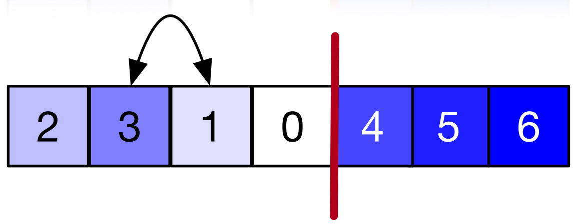
- 

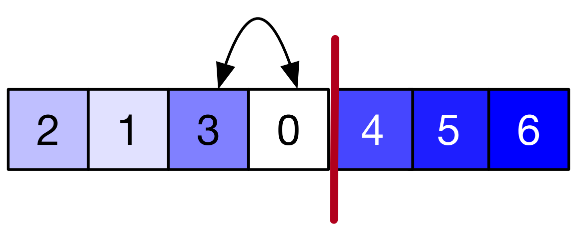
- Third largest element has bubbled up to the correct place

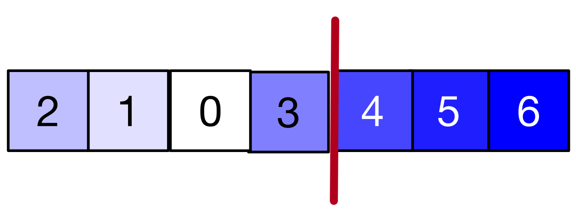
- 

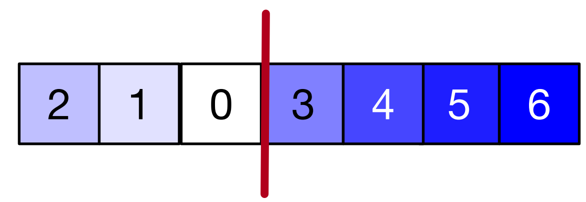
Bubble Sort

- Fourth pass
  - 

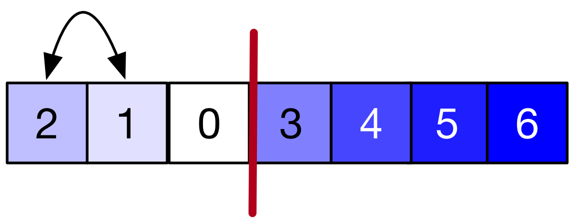
  - 

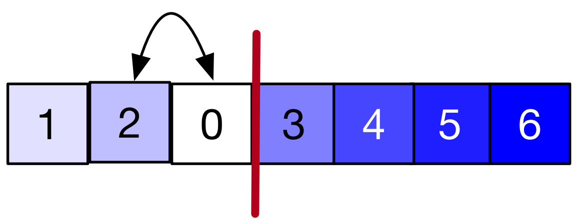
  - 

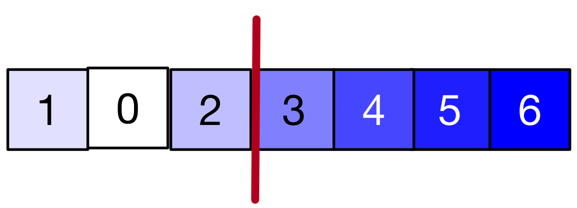
  - 

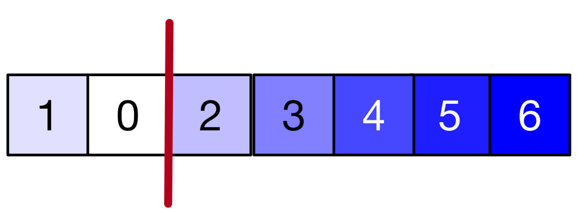
- Now 3 has bubbled up
  - 

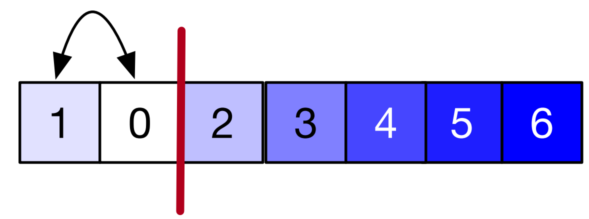
Bubble Sort

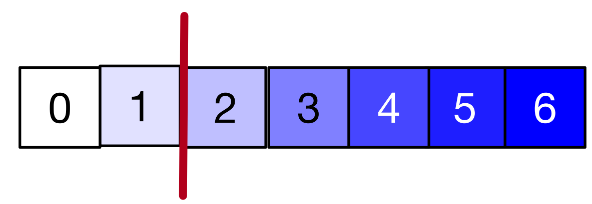
- Fifth Pass
  - 

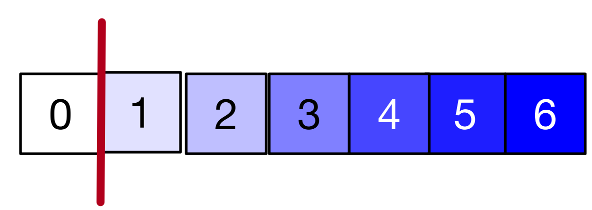
  - 

  - 

- 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

Bubble Sort Invariant

- Loop Invariant:
  - After execution of outer loop with value `i`
    - `arr[n-i-1:n]` contains the `i+1` maximal values in ascending order

```
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 Invariant

- After execution of outer loop with value `i`
  - `arr[n-i-1:n]` contains the `i+1` maximal values in ascending order
  - Follows from:
    - The inner loop selects the maximum element in `arr[0:n-i]` and moves it to `arr[n-i-1]`

```
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]
```

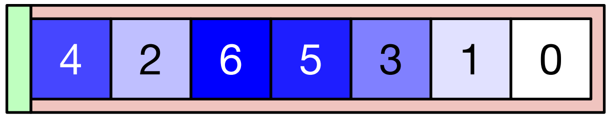
Insertion Sort

Thomas Schwarz, SJ

Insertion Sort

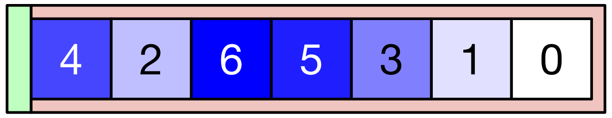
- Idea:
  - Break the array into a sorted and an unsorted part
    - Move first element of the unsorted part into the correct position in the sorted array

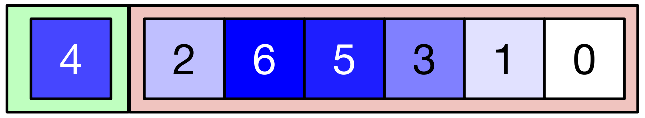
Insertion Sort

- Example:
  - Sort 

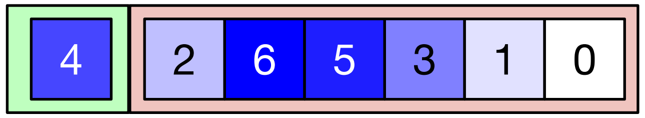
    - Reddish part is unsorted: initially whole array
    - Greenish part is sorted: initially empty

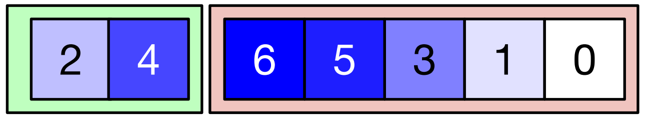
Insertion Sort

- Example:
  - 

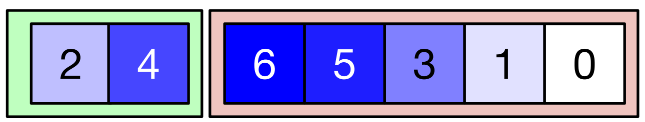
  - First element in the red part is 4:
  - Insert 4 into the green part
  - 

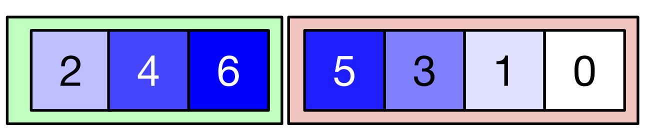
Insertion Sort

- Example
  - 

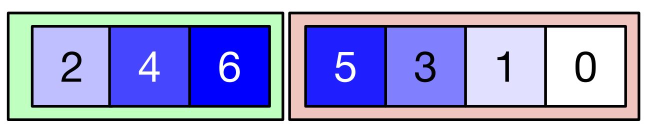
  - Next unsorted element is 2
  - Compare with 4
  - Insert in front of 4
  - 

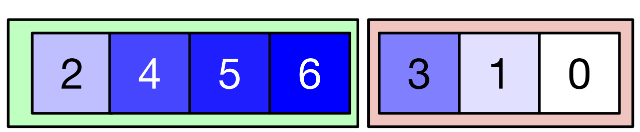
Insertion Sort

- Example
  - 

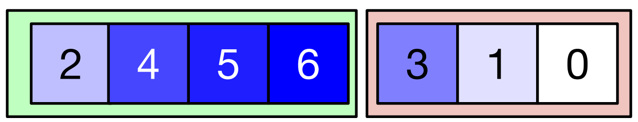
  - Next unsorted element is 6
    - Compare with 2, then 4
    - Insert after 4
  - 

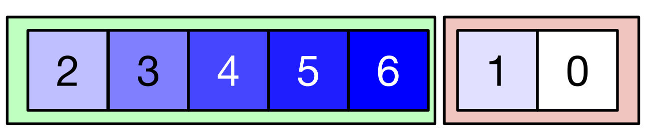
Insertion Sort

- Example
  - 

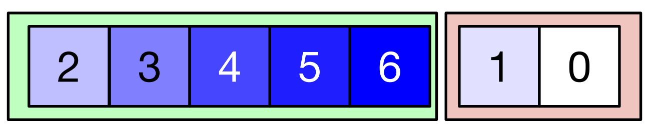
  - Next unsorted element is 5
  - Compare with 2, 4, 6
    - Insert before 6
  - 

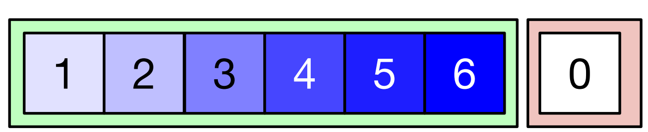
Insertion Sort

- Example
  - 

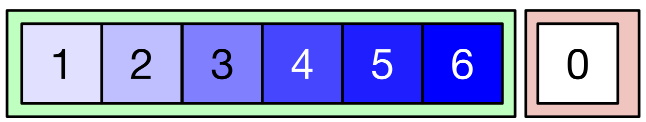
  - Next unsorted element is 3
  - Compare with 2, then 4
  - Insert before 4
  - 

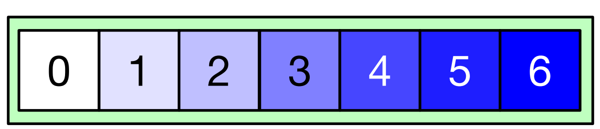
Insertion Sort

- 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

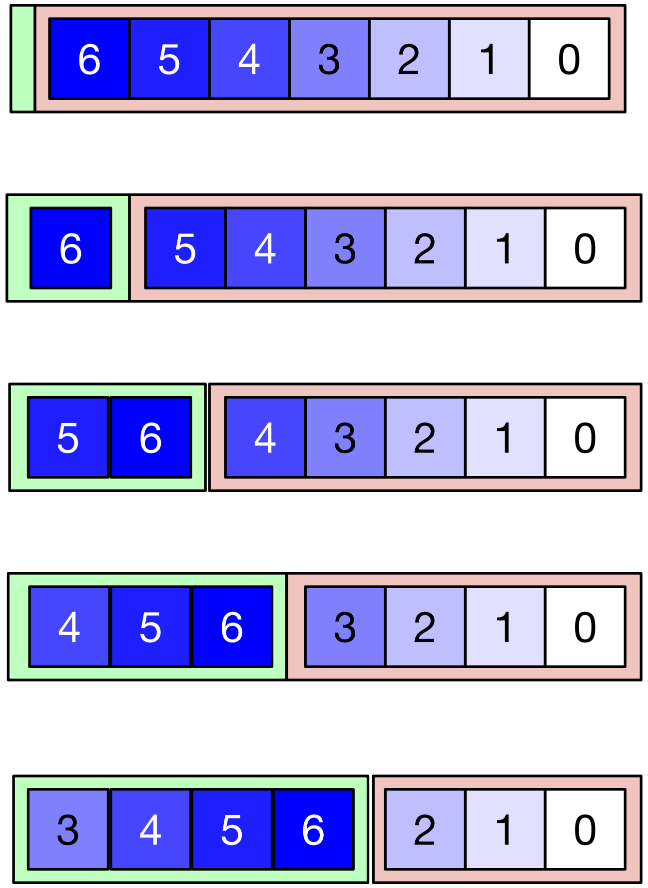
- Performance:
  - Inserting at a specific index in an array means moving the elements after the insertion
    - This is a big hidden cost
  - Inserting at a specific index into a linked list only involves finding the insertion point and constant link resetting work
  - However, we can now avoid comparisons
  - 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



Insertion Sort Invariant

- After each step, the green array is correctly sorted
- After each step, the multi-set of elements has not changed