Types of Algorithms

Mohammad Ghoddosi

Types of Algorithms

- Brute force Algorithm
- Recursive Algorithm
- Backtracking Algorithm
- Divide and Conquer Algorithm
- Greedy Algorithm
- Dynamic Programming Algorithm

- Searching Algorithm
- Sorting Algorithm

Brute force Algorithm

- Check all possible ways
- All possible solution
- Pros
  - Guaranteed way to find the correct solution
  - Ideal for solving small and simpler problems
- Cons
  - Inefficient
  - Often goes above $O(n!)$

- Find smallest and largest n digit numbers with sum digits equal to k

Recursive Algorithm

- Function calls itself directly or indirectly
- Calling a copy of itself and solving smaller sub-problems
  - Breaking the problem into sub problems of the same type
  - Base condition
- Properties of Recursion
  - Performing the same operations multiple times with different inputs
  - In every step we solve smaller inputs and smaller problem
  - Base condition to stop recursion
- Factorial, Fibonacci, Tower of Hanoi, DFS, …

Backtracking Algorithm

- Solving the problem in an incremental way
  - Recursively
- Build solution one piece at a time
- Remove solutions that fail to satisfy the constraints

- Hamiltonian Cycle
- Sudoku
- N queen problem

Divide and Conquer Algorithm

- Solve the problem in three steps
  - Divide: Divide the problem into sub problems
  - Conquer: Solve sub problems independently
  - Combine: Add the combined results

Greedy Algorithm

- Solution is built part by part
- Based on immediate benefit
- Make local optimal choice and hope of finding a global optimum
- Problem properties:
  - Greedy choice property: Choosing the best solution in each phase leads to global optimal
  - Optimal substructure: optimal solution contains the optimal solution to the subproblems

- Dijkstra, Prim, Kruskal, …

Example for greedy proof (prim)

- Greedy solution is always part of some optimum solution
  - T' is a subtree of a MST for G. then there is a complete MST for G that contains T' and contains the edge (u, v) that is the minimum weight edge from a vertex in T' to a vertex not in T'
  - Consider T* (MST with T' and without (u, v)
  - Consider the unique path between u and v
  - The unique path leaves T' with some edge (y, z)
    - y is in T' and z isn't
  - Remove (y, z) and add (u, v)
    - New graph is a tree
    - weight (u, v) <= weight (y, z)

Dynamic Programming Algorithm

- Optimization over recursion
  - Recursion with repeated calls for the same input
- Store the results of sub problems
- Prevent re computing the same solution
- Reduces time complexity
  - Recursive Fibonacci: $O(2^n)$
  - DP Fibonacci: $O(n)$

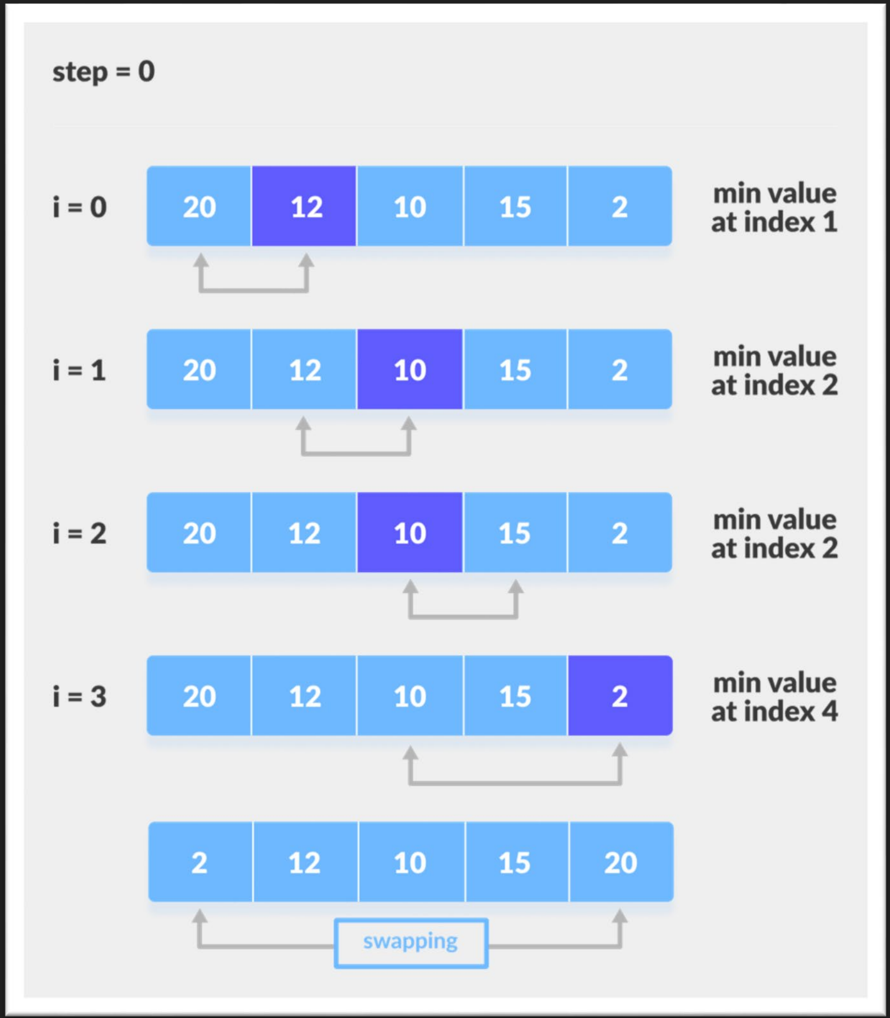
Sorting algorithms

- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Heap sort
- Counting sort
- …

Selection sort

- Each time find the current minimum
- Swap minimum with first element
- Time complexity: $O(n^2)$

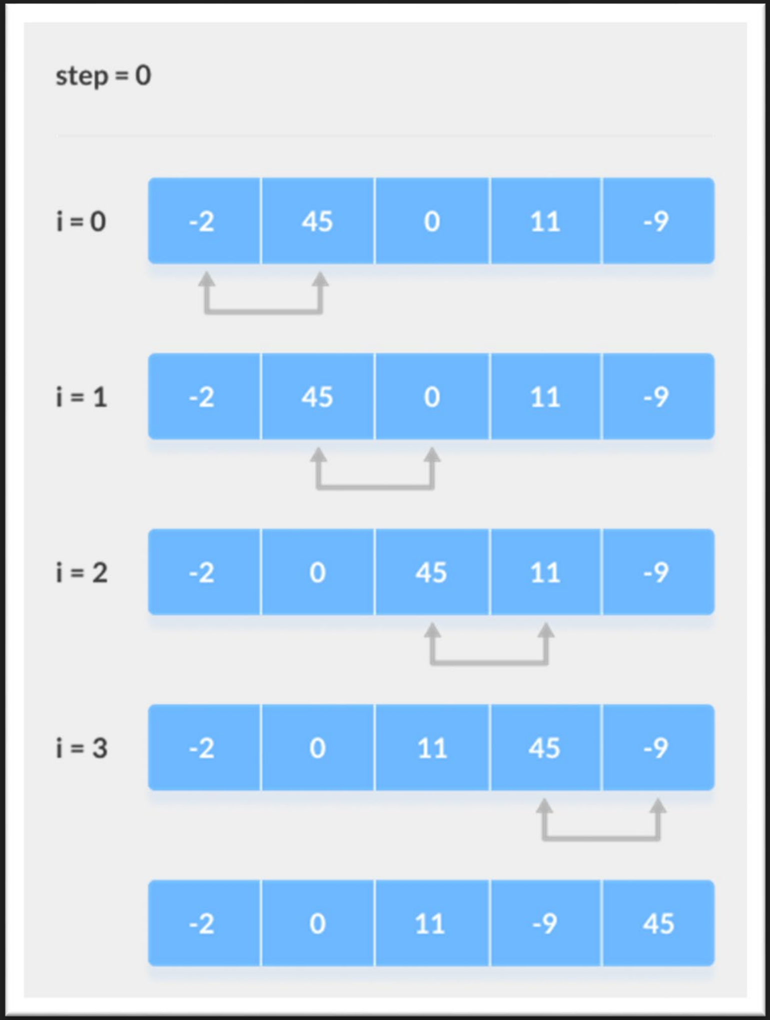
```
for i in range(len(A)):
    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i + 1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    A[i], A[min_idx] = A[min_idx], A[i]
```



Bubble sort

- Compare adjacent elements
- Swap if the order is incorrect
- Time complexity: $O(n^2)$

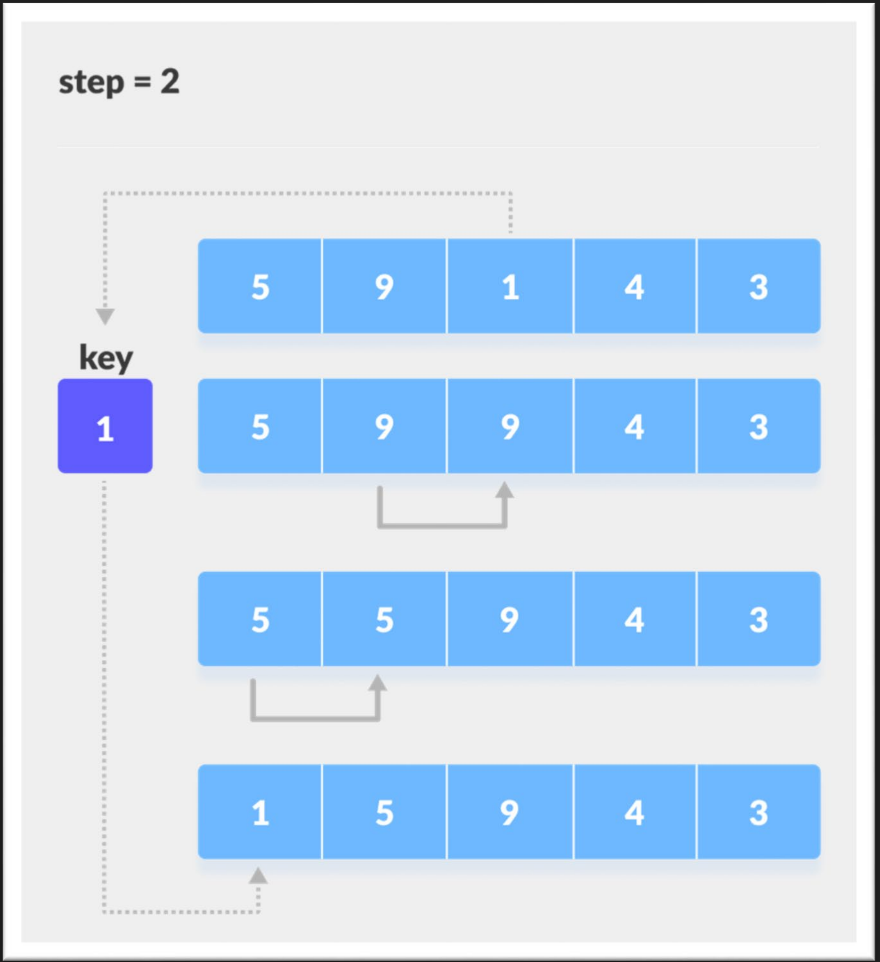
```
for i in range(len(A)):
    for j in range(0, len(A) - i - 1):
        if A[j] > A[j + 1]:
            A[j], A[j + 1] = A[j + 1], A[j]
```



Insertion sort

- Each time insert I’th element in sorted first part of the array
- Finds new data location and moves data to its place
- Shift all other cells
- Time complexity: $O(n^2)$

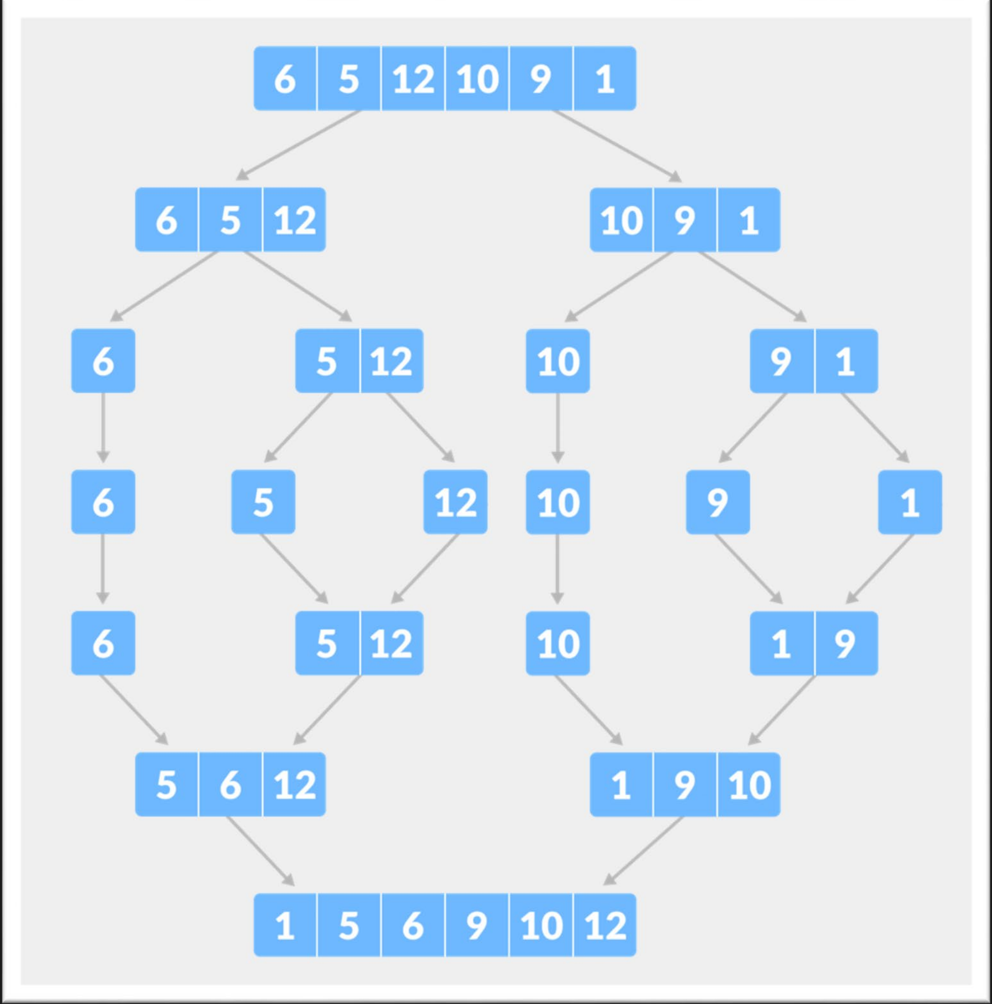
```
for step in range(1, len(A)):
    key = A[step]
    j = step - 1

    while j >= 0 and key < A[j]:
        A[j + 1] = A[j]
        j = j - 1

    A[j + 1] = key
```



Merge sort

- Split array into two halves
- Sort each half recursively
- Merge sorted arrays
- Time complexity: $O(n \log n)$

```python
def mergeSort(array):
    if len(array) <= 1:
        return array

    mid = len(array)//2

    L = mergeSort(array[:mid])
    R = mergeSort(array[mid:])
    return combine_sorted(L, R)
```



Merge sort

```python
def mergeSort(array):
    if len(array) <= 1:
        return array

    mid = len(array)//2

    L = mergeSort(array[:mid])
    R = mergeSort(array[mid:])
    return combine_sorted(L, R)
```

```python
def combine_sorted(L, R):
    i = j = 0
    array = []
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            array.append(L[i])
            i += 1
        else:
            array.append(R[j])
            j += 1

    return array + L[i:] + R[j:]
```

Counting sort

- Not based on camparison
- Find the maximum element
- Generate a new list of zeros with len = max(Array)
- Count each value
- Time complexity = $O(n+m)$
- Linear complexity
- More memory usage

```python
def countingSort(array):
    count = [0] * max(array)
    res = []

    for value in array:
        count[value] += 1

    for i in range(len(count)):
        res = res + ([i] * count[i])
```

honorable mentions

- Quick sort
  - Divide the array based on a pivot
- Pancake sort
  - Sort only by flipping array from 0 to I
  - Find max, flip twice to place it at the end
- Bogo sort
  - Random permutation until success
- Sleep sort
  - Generate new thread for each value and sleep it for time = value
  - Print the value of each thread after sleep