m2_ICA

June 3, 2025

0.1 Module 2

0.1.1 ICA: Searches

Robert Clay Harris: jbm2rt

ICA 1: Linear Search

[3]:
```python
arr = [23, 1, 45, 34, 17]
tgt = 34

def search_linear(arr, tgt):
    n = len(arr)
    for i in range(n):
        if arr[i] == tgt:
            return i
    return -1

search_linear(arr,tgt)
```

[3]: 3

```
pros:
    - simplicity
cons:
    - no concept of efficiency
time complexity: best O(1), worst O(n)
when to use?
    - when the array is small in size and the burden of computation is small
    - when you don't need to search often or with complexity
```

ICA 2: Binary Search

[15]:
```python
arr = [1, 4, 7, 9, 15, 24, 30]
tgt = 30

def search_binary(arr, tgt):
    # st, end <- arr[0], arr[-1]
    # compute mid between start and end
    # arr[mid] == tgt -> return mid
    # arr[mid] < tgt -> end = mid
```

```python
    # arr[mid] > tgt -> st = mid

    n = len(arr)
    st, end = 0, n-1
    if arr[st]  == tgt: return st
    if arr[end] == tgt: return end
    while True:
        mid = ((end - st) // 2) + st

        if mid == end or mid == st:
            return -1

        if arr[mid] == tgt: return mid
        elif arr[mid] >= tgt: end = mid
        else:
            st = mid

search_binary(arr, tgt)
```

[15]: 6

```
pros:
    - much faster than linear search
cons:
    - requires sorted array
    - more complex to implement than linear search
time complexity: best O(1), worst O(log n)
when to use?
    - when the array is large and sorted
    - when you need to search often or with complexity
```

ICA 3: Sort Selection

[ ]:
```python
arr = [34, 17, 23, 1, 45]

def sort_selection(arr):
    # i: 0~n-1
        # j: i ~ n

    n = len(arr)
    for i in range(0, n-1):
        min_val = arr[i]
        min_idx = i
        for j in range(i, n):
            if arr[j] < min_val:
                min_val = arr[j]
                min_idx = j
```

```
        temp = arr[i]
        arr[i] = min_val
        arr[min_idx] = temp
    return arr

sort_selection(arr)
```

[ ]: [1, 17, 23, 34, 45]

```
pros:
    - easy to implement
    - no additional data structures needed
    - works well for small datasets
cons:
    - not efficient for large datasets
    - can be slow for large arrays
time complexity: best O(n^2), worst O(n^2)
when to use?
    - when the array is small and unsorted
    - when you need to search for an element in a small dataset
```

ICA 4: Quick Sort

[22]:
```
arr = [12, 4, 5, 6, 7, 3, 1, 15]

def sort_quick(arr):
    if len(arr) <= 1:
        return arr
    pivot_idx = len(arr) // 2

    left_arr, right_arr = divide(arr, pivot_idx)
    sorted_left = sort_quick(left_arr)
    sorted_right = sort_quick(right_arr)

    return sorted_left + [arr[pivot_idx]] + sorted_right

def divide(arr, pivot_idx):
    arr_left = []
    arr_right = []
    n = len(arr)
    for i in range(n):
        if i == pivot_idx:
            continue
        if arr[i] <= arr[pivot_idx]:
            arr_left.append(arr[i])
        else:
            arr_right.append(arr[i])
```

```python
    return arr_left, arr_right

sort_quick(arr)
```

[22]: [1, 3, 4, 5, 6, 7, 12, 15]

```
pros:
    - easy to implement
    - no additional data structures needed
cons:
time complexity: best O(n log n), worst O(n^2), avg O(n log n)
when to use?
    - if your dataset is large and unsorted
    - when you need to sort the array for further operations
```

ICA 5: Shell Sort description:
- Pick a sequence of gaps (for example, n/2, n/4, …, 1)
- For each gap, run an insertion pass on the array, comparing and swapping elements that are the defined gap apart
- Cut the gap in half (or to the next sequence value) and repeat until the gap is 1
- Final pass with gap = 1 finishes the sort

pros:
- Large gap passes move elements long distances early so by the time the final pass runs the array is already mostly sorted and requires fewer shifts

cons:
- Performance depends on the chosen gap sequence
- Finding an optimal sequence can be complex and may not yield the best results for all datasets

time complexity:
- Worst case scenario: $O(n^2)$
- Best case scenario: $O(n \log n)$
- Each gap pass n elements $O(n)$, and there are about $O(\log n)$ passes depending on gaps

when to use?
- When working with moderate-sized or partially sorted data and need in-place method faster than n^2
- Useful when memory is limiting factor