Fast Sorting in NumPy

sorting arrays in numpy.
Fast Sorting in NumPy: np.sort and np.argsort

Sorting arrays efficiently in NumPy can be done using **`np.sort()`** and **`np.argsort()`**. Here's a breakdown of both:

1. **`np.sort()`** - Direct Sorting

- Returns a sorted copy of the array.
- Default is ascending order, but you can reverse it.
- Works on multi-dimensional arrays with an axis parameter.

Example:

```
python

import numpy as np

arr = np.array([3, 1, 5, 2, 4])
sorted_arr = np.sort(arr)  # Ascending order
sorted_arr_desc = np.sort(arr)[::-1]  # Descending order

print(sorted_arr)        # [1 2 3 4 5]
print(sorted_arr_desc)   # [5 4 3 2 1]
```

2. **`np.argsort()`** - Indices of Sorted Elements

- Returns indices that would sort the array.
- Useful for sorting one array based on another.

Example:

```
python

arr = np.array([3, 1, 5, 2, 4])
sorted_indices = np.argsort(arr)

print(sorted_indices)    # [1 3 0 4 2]
print(arr[sorted_indices])  # [1 2 3 4 5]
```

Sorting Along an Axis in a 2D Array

```
python

matrix = np.array([[3, 2, 1], [6, 5, 4]])
sorted_matrix = np.sort(matrix, axis=1)  # Sort each row
```

```
print(sorted_matrix)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

Fast Sorting Algorithms in NumPy

NumPy uses quicksort by default but allows other algorithms:

- `'quicksort'` (default, fast for general cases)
- `'mergesort'` (stable sort, better for partially sorted data)
- `'heapsort'` (useful for worst-case scenarios)
- `'stable'` (like mergesort but keeps order of equal elements)

Example of Specifying Algorithm

```
python

arr = np.array([3, 1, 5, 2, 4])
sorted_arr = np.sort(arr, kind='mergesort')
print(sorted_arr)  # [1 2 3 4 5]
```

Would you like a performance comparison of these sorting algorithms? 🚀

go on!
(also include following topics: Partial Sorts: Partitioning, example: k-nearest neighbours)

Fast Sorting in NumPy: Performance & Special Techniques 🚀

NumPy provides **multiple sorting techniques** for different use cases, including **full sorts, partial sorts, and optimized sorting algorithms**. Let's dive deeper into **performance comparisons** and **specialized sorting** techniques like **partial sorting for k-nearest neighbors**.

---

1. Performance Comparison of Sorting Algorithms

NumPy supports different sorting algorithms:

- `quicksort` (default) → Fastest but unstable.
- `mergesort` → Slower but stable.
- `heapsort` → Good for worst-case but generally slower.
- `stable` → Ensures equal elements maintain their order.

Performance Test

Let's test their speed on a large array:

```python
import numpy as np
import time

arr = np.random.randint(0, 10000, 10**6)  # Large array with 1 million elements

for algo in ["quicksort", "mergesort", "heapsort", "stable"]:
    start = time.time()
    np.sort(arr, kind=algo)
    end = time.time()
    print(f"{algo}: {end - start:.5f} seconds")
```

📌 **Expected Outcome:** `quicksort` is usually the fastest, while `mergesort` and `stable` are slower but useful for specific cases.

---

2. Partial Sorting: Partitioning

Sometimes, we **don't need to fully sort an array**, just find the **smallest/largest k elements** efficiently.

Using `np.partition()` for k-smallest or k-largest elements

- **Faster than full sorting**
- Uses **introselect** (hybrid of quicksort & heapsort)
- Time complexity: **O(n)** (instead of O(n log n) for full sort)

Example: Find the Top-3 Smallest Elements

```python
arr = np.array([7, 2, 9, 4, 1, 5, 6, 3, 8])

k = 3  # We want the 3 smallest elements
partitioned_arr = np.partition(arr, k)  # Places k-smallest in first k positions

print(partitioned_arr[:k])  # [1 2 3] (unordered but correct elements)
```

Finding the k-Largest Elements

To find the **top-k largest**, use `-k`:

```python
print(np.partition(arr, -k)[-k:])  # [7 8 9]
```

---

3. Example: k-Nearest Neighbors (KNN)

The **k-nearest neighbors (KNN)** algorithm often requires finding **k closest distances** efficiently.

Example: Finding 3 Nearest Neighbors

```
python


points = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])  # 4 points in 2D space
target = np.array([4, 5])  # Our reference point

# Compute Euclidean distances
distances = np.linalg.norm(points - target, axis=1)

# Find 3 nearest neighbors using np.argpartition()
k = 3
nearest_indices = np.argpartition(distances, k)[:k]

print("Nearest neighbor indices:", nearest_indices)
print("Nearest points:", points[nearest_indices])
```

🚀 **Efficiency Boost:**
Using `np.argpartition()` instead of `np.argsort()` improves performance **from O(n log n) to O(n)**!

Key Takeaways

✅ **Use `np.sort()` for full sorting** (choose algorithm based on stability needs).
✅ **Use `np.argsort()` to get indices of sorted elements**.
✅ **Use `np.partition()` for faster selection of top-k elements** (useful in KNN & top-k problems).
✅ **For large datasets, avoid full sorting—partitioning is much faster!**

⚡ Need help optimizing a specific problem? Let me know! 🚀