MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Algorithm Analysis

Laboratory work 2 : Study and empirical analysis of sorting algorithms.

Elaborated:
st.gr. FAF-211 Gazea Sandu
Verified:
asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Sorting algorithms are a fundamental concept in computer science and are essential for organizing and processing data efficiently. In computer science, sorting refers to the process of rearranging a collection of items in a specific order. Sorting algorithms are used to sort data structures like arrays, lists, and trees, and the efficiency of the sorting algorithm is determined by the number of comparisons and swaps it takes to sort the data.

There are numerous sorting algorithms, each with its own strengths and weaknesses. Some of the most popular sorting algorithms include bubble sort, selection sort, insertion sort, merge sort, quicksort, and heapsort. Each algorithm has its own approach to sorting data, with different time and space complexity.

Understanding sorting algorithms is crucial for computer scientists, as efficient sorting can improve the performance of various applications such as databases, search engines, and operating systems. Moreover, a deeper understanding of sorting algorithms can help you become a better programmer and problem-solver, regardless of the field you're in. Sorting algorithms can be categorized as either internal or external, depending on how the data is stored during the sorting process. Internal sorting algorithms sort data that can fit into the main memory of a computer, while external sorting algorithms are used for data that is too large to fit into main memory and must be sorted on disk or other external storage media.

Efficient sorting algorithms are vital for handling large datasets, where the time and space complexity of the algorithm can make a significant impact on the performance. There are various ways to measure the efficiency of sorting algorithms, including the number of comparisons and swaps, the time taken to sort the data, and the amount of memory used.

Sorting algorithms also have numerous real-world applications, from sorting names in a phone book to sorting large datasets in scientific research. They are used in various industries, including finance, healthcare, and engineering. Additionally, sorting algorithms are often used in combination with other algorithms to optimize complex operations, such as data compression and searching.

Overall, sorting algorithms play a crucial role in computer science and data processing. Understanding how they work and their different approaches can help you choose the most efficient algorithm for a given task, optimize the performance of your code, and improve your problem-solving skills.

QuickSort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time. Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort.[4] The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Most implementations of quicksort are not stable, meaning that the relative order of equal sort items is not preserved.

Mathematical analysis of quicksort shows that, on average, the algorithm takes O($n \log n$) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons.

Code:

```
def partition(array, low, high):

# Choose the rightmost element as pivot
pivot = array[high]

# Pointer for greater element
i = low - 1

# Traverse through all elements
# compare each element with pivot
for j in range(low, high):
```

```
            if array[j] <= pivot:
                # If element smaller than pivot is found
                # swap it with the greater element pointed by i
                i = i + 1

                # Swapping element at i with element at j
                (array[i], array[j]) = (array[j], array[i])

        # Swap the pivot element with
        # e greater element specified by i
        (array[i + 1], array[high]) = (array[high], array[i + 1])

        # Return the position from where partition is done
        return i + 1

# Function to perform quicksort


def quick_sort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quick_sort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quick_sort(array, pi + 1, high)
```

```
array = [10, 7, 8, 9, 1, 5]
quick_sort(array, 0, len(array) - 1)

print(f'Sorted array: {array}')
```

Mergesort

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Code:

```
def mergeSort(arr):
if len(arr) > 1:

     # Finding the mid of the array
    mid = len(arr)//2

    # Dividing the array elements
    L = arr[:mid]
```

```
        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Code to print the list
```

```
def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()



if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)
```

HeapSort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

The heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

What is meant by Heapify?

Heapify is the process of creating a heap data structure from a binary tree represented using an array. It is used to create Min-Heap or Max-heap. Start from the last index of the non-leaf node whose index is given by n/2 – 1. Heapify uses recursion.

Code:

```
def heapify(arr, N, i):
largest = i  # Initialize largest as root
l = 2 * i + 1     # left = 2*i + 1
```

```
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < N and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < N and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap

        # Heapify the root.
        heapify(arr, N, largest)

# The main function to sort an array of given size


def heapSort(arr):
    N = len(arr)

    # Build a maxheap.
    for i in range(N//2 - 1, -1, -1):
        heapify(arr, N, i)

    # One by one extract elements
    for i in range(N-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        heapify(arr, i, 0)
```

```
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]

    # Function call
    heapSort(arr)
    N = len(arr)

    print("Sorted array is")
    for i in range(N):
        print("%d" % arr[i], end=" ")
```

Insertion Sort

Insertion sort is a simple sorting algorithm that sorts an array of elements by gradually building a final sorted array. It is an in-place comparison-based sorting algorithm that works by iterating through an array and inserting each element into its proper position in a sorted subarray.

The insertion sort algorithm works by starting with the second element of the array, and comparing it with the first element. If the second element is smaller than the first, the two elements are swapped. The algorithm then moves on to the third element, comparing it with the second element, then the first, and swapping if necessary to ensure that the first three elements are in sorted order. The algorithm continues in this way, iterating through the entire array, and inserting each element into its correct position in the sorted subarray.

The worst-case time complexity of insertion sort is O(n*n), where n is the number of elements in the array. This occurs when the array is in reverse order, so that each element must be compared and swapped with every other element in the array. The best-case time complexity of insertion sort is O(n), which occurs when the array is already sorted. The average time complexity of insertion sort is also O(n*n).

Insertion sort has the advantage of being simple and easy to implement, and it works well for small arrays or partially sorted arrays. However, it is not well-suited for large arrays, as its time complexity is relatively slow.

```
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key


arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for i in range(len(arr)):
    print ("% d" % arr[i])
```

Implementation

Code in python:

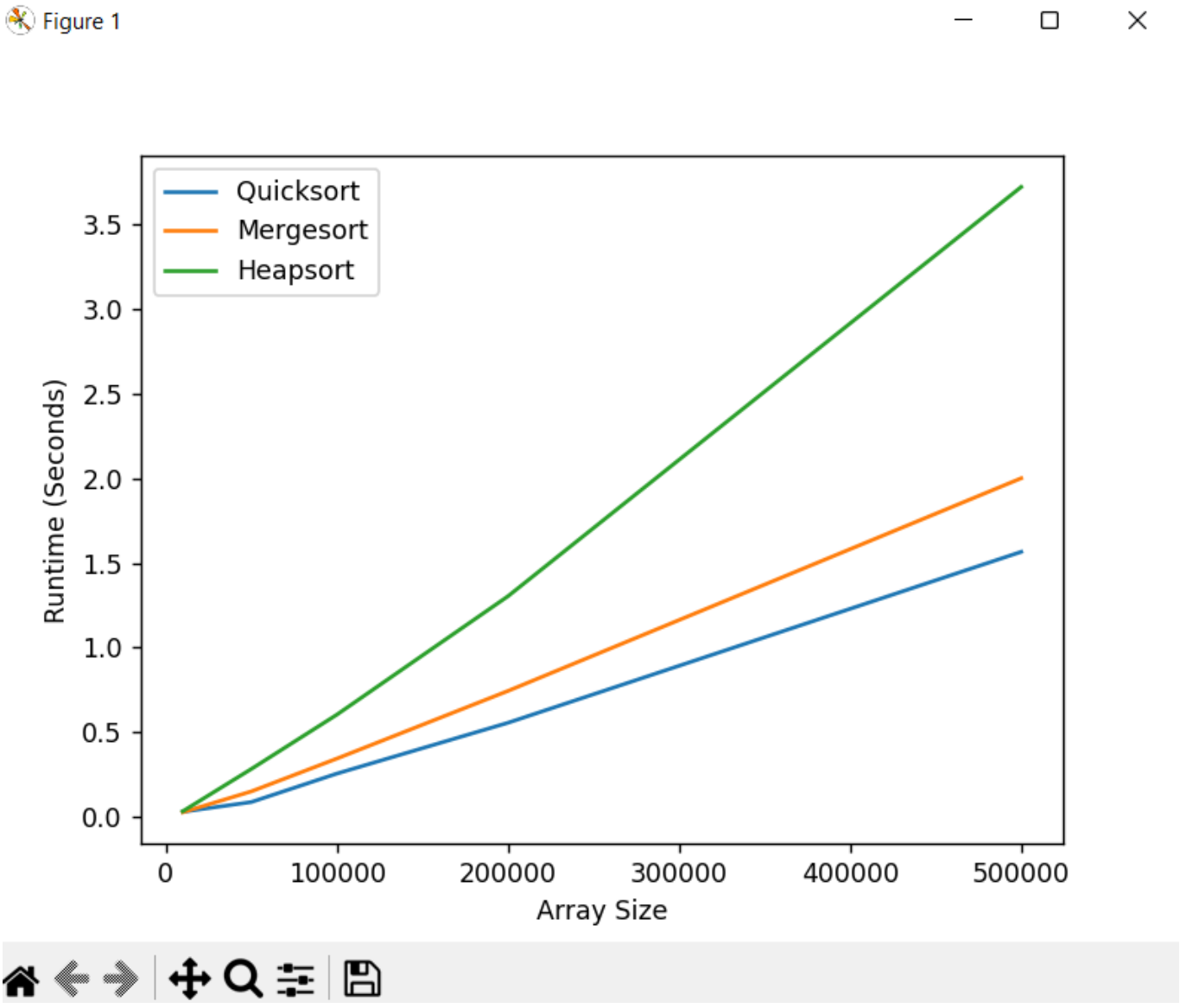
```
import time
import random
import matplotlib.pyplot as plt



def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x < pivot]
    right = [x for x in arr[1:] if x >= pivot]
    return quicksort(left) + [pivot] + quicksort(right)

def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergesort(arr[:mid])
    right = mergesort(arr[mid:])
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
```

```
    return result

def heapify(arr, n, i):
    largest = i
    left = 2*i + 1
    right = 2*i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapsort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

def insertionsort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Generate random arrays to sort
sizes = [10000, 50000, 100000, 200000, 500000]
```

```
arrays = {}
for size in sizes:
    arrays[size] = [random.randint(1, size) for _ in range(size)]

# Sort the arrays with each algorithm and record the runtimes
quicksort_times = []
mergesort_times = []
heapsort_times = []
insertionsort_times = []

for size in sizes:
    array = arrays[size]
    start_time = time.time()
    quicksort(array)
    end_time = time.time()
    quicksort_times.append(end_time - start_time)

    array = arrays[size]
    start_time = time.time()
    mergesort(array)
    end_time = time.time()
    mergesort_times.append(end_time - start_time)

    array = arrays[size]
    start_time = time.time()
    heapsort(array)
    end_time = time.time()
    heapsort_times.append(end_time - start_time)

    array = arrays[size]
    start_time = time.time()
    insertionsort(array)
    end_time = time.time()
    insertionsort_times.append(end_time - start_time)
```

```
plt.plot(sizes, quicksort_times, label="Quicksort")
plt.plot(sizes, mergesort_times, label="Mergesort")
plt.plot(sizes, heapsort_times, label="Heapsort")
plt.plot(sizes, insertionsort_times, label="insertionsort")
plt.xlabel("Array Size")
plt.ylabel("Runtime (Seconds)")
plt.legend()
plt.show()
```

Screenshot:



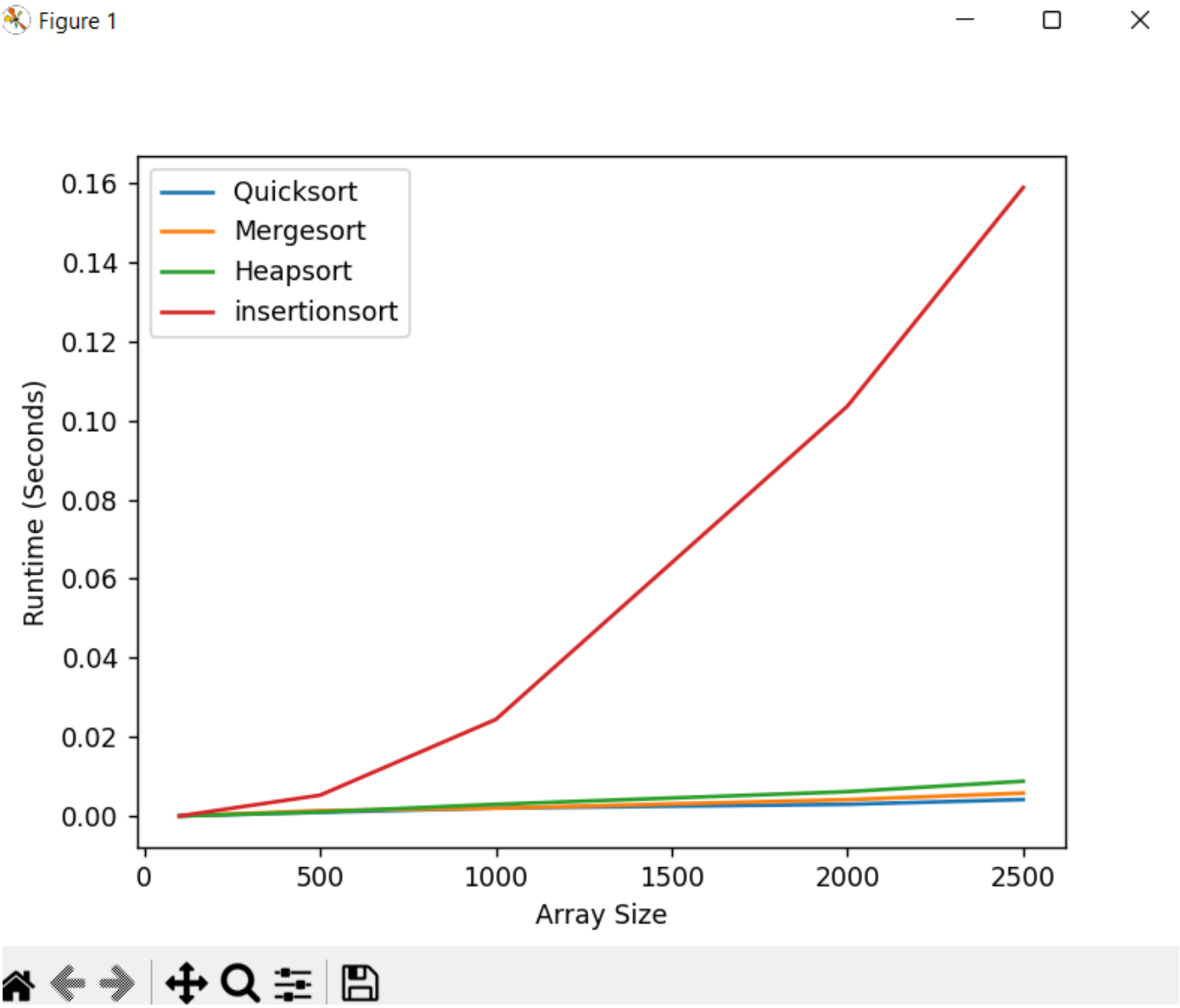


Figure 1
Quicksort
Mergesort
Heapsort
insertionsort
Runtime (Seconds)
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
500
1000
1500
2000
2500
Array Size

Conclusion

To summarize, the performance of quicksort, mergesort, heapsort, and insertion sort when sorting an array of 500,000 elements is subject to several factors, including the input data, hardware and software environment, and implementation details. In practice, quicksort and mergesort tend to perform well with an average time complexity of O(n log n), with quicksort being preferable due to its better cache locality and lower constant factors. Heapsort is useful when stable sorting is not necessary and space complexity is a concern, with a worst-case time complexity of O(n log n). Insertion sort is straightforward and ideal for small or partially sorted arrays, but its time complexity is comparatively slow for large arrays.

For sorting large arrays, quicksort is a common and reliable choice, although mergesort and heapsort may also be suitable depending on specific needs and limitations. It's best to test and evaluate different algorithms with your actual data and hardware setup to make an informed decision. Overall, when selecting a sorting algorithm, consider factors such as the input data, hardware and software environment, and requirements of your application. Quicksort, mergesort, and heapsort are generally efficient and dependable for sorting large datasets, while insertion sort may be appropriate for specialized use cases or smaller arrays.