DS 7^{th} homework

58121102 Jiang Yuchu

December 16, 2022

P_{401} 3. Write a C++ function that implements Linked Insertion Sort. What is the worst-case number of comparisons made by your sort function? What is the worst-case number of record moves made? How do these compare with the corresponding numbers for Program 7.5?

Answer:

See *p401-3.cpp*. I implemented the insertion sort on singly linked list.

In the worst case, this code's insertion sort does $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$ comparisons, where n is the number of nodes in the linked list. At the same time, it will also move $\frac{n(n-1)}{2}$ records. In the worst case, this code has a time complexity of $O(n^2)$, which is same as Program 7.5.

P_{405} 5. Quick Sort is an unstable sorting method Give an example of an input list in which the order of records with equal keys is not preserved.

Answer:

Consider the following example:

$$\text{Input:} \quad [5, 4, 9, 5, 3, 1, 7, 5]$$

The Quick Sort algorithm first selects a pivot element (in this case, let's say the first element, 5, is selected as the pivot), and arranges the other elements in the list into two partitions: those that are less than or equal to the pivot, and those that are greater than the pivot. The first partition would be [4, 3, 1], and the second partition would be [9, 7]. Then, the algorithm would recursively sort the two partitions, which would result in the following list:

$$\text{Sorted:} \quad [1, 3, 4, 5, 7, 9, 5, 5]$$

As you can see, the order of the two 5s at the end of the list is not preserved, which means that the Quick Sort algorithm is an unstable sorting method.

P_{412} 1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of each phase of MergeSort (Program 7.9).

Answer:

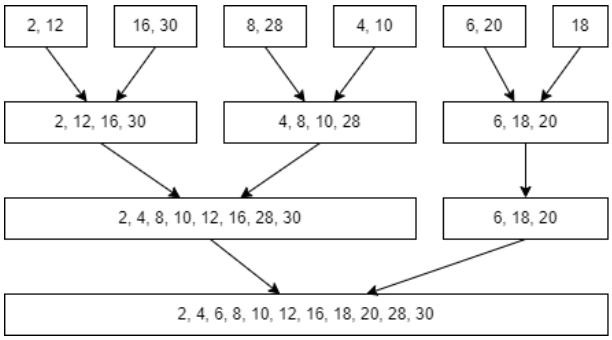


Figure 1: Status at the end of each phase of MergeSort

See figure 1.

P_{416} 2. Heap Sort is unstable. Give an example of an input list in which the order of records with equal keys is not preserved.

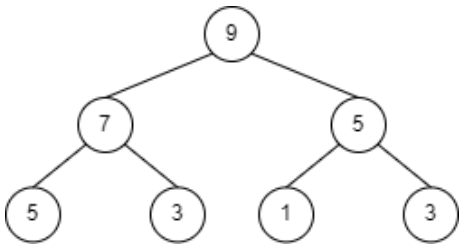
Answer:

Consider the following example:

$$\text{Input:} \quad [5, 4, 9, 5, 3, 1, 7, 5]$$

The Heap Sort algorithm first builds a max-heap from the input list, where the largest element is at the root of the heap. Then, it repeatedly removes the root element and adds it to the end of the list, rebuilding the heap each time.

The heap-building phase would result in the following max-heap:



The first element removed from the heap would be 9, followed by 7, 5, 5, 3, 1, and 4, in that order. After removing all of the elements from the heap, the final sorted list would be:

$$\text{Sorted:} \quad [1, 3, 4, 5, 5, 5, 7, 9]$$

As you can see, the order of the two 5s in the middle of the list is not preserved, which means that the Heap Sort algorithm is an unstable sorting method.

P_{422} 5. If we have n records with integer keys in the range $[0, n^2)$, then they may be sorted in $O(nlogn)$ time using Heap Sort or Merge Sort. Radix Sort on a single key (i.e., $d = 1$ and $r = n^2$) takes $O(n)$ time. Show how to interpret the keys as two subkeys so that Radix Sort will take only $O(n)$ time to sort n records. (Hint: Each key, K_i, may be written as $K_i = K_i^1 n + K_i^2$ with K_i^1 and K_i^2 integers in the range $[0, n)$.

Answer:

To sort n records with integer keys in the range $[0, n^2)$ using Radix Sort in $O(n)$ time, we can interpret the keys as two subkeys. Each key, K_i, can be written as $K_i = K_i^1 n + K_i^2$, where K_i^1 and K_i^2 are integers in the range $[0, n)$.

For example, if we have the key $K_i = 53$, we can interpret it as the two subkeys $K_i^1 = 5$ and $K_i^2 = 3$ with $n = 10$. Similarly, if we have the key $K_i = 150$, we can interpret it as $K_i^1 = 7$ and $K_i^2 = 10$ with $n = 20$.

To sort the records using Radix Sort, we can first sort the records based on their K_i^1 subkeys using the least significant digit sorting method. This will result in a list of records that are partially sorted based on their K_i^1 subkeys.

Next, we can sort the records based on their K_i^2 subkeys using the least significant digit sorting method. This will result in a list of records that are fully sorted based on their K_i^1 and K_i^2 subkeys.

Since the range of the subkeys K_i^1 and K_i^2 is $[0, n)$, the number of digits in each subkey is $lg(n)$, which means that the total number of digits in each key is $2lg(n)$. Since the Radix Sort algorithm sorts the records in $O(n)$ time for each digit, the overall time complexity of the sorting process is $O(n * 2 * lg(n)) = O(n)$.

Experiment

p_{435} 5.

1 Experiment Goal

1.1 Basic Tasks

The objective of this assignment is to come up with a composite sorting function that is good on the worst-time criterion. The candidate sort methods are (a) Insertion Sort (b) Quick Sort (c) Merge Sort

(d) HeapSort

To begin with, program these sort methods in C++ in each case, assume that n integers are to be sorted.In the case of Quick Sort, use the median-of-three method. In the case of MergeSort use the iterative method. Check out the correctness of the programs using some test data.

To obtain reasonably accurate runtimes, you need to know the accuracy of the clock or timer you are using. Determine this by reading the appropriate manual. Let the clock accuracy be δ. Now, run a pilot test to determine ballpark times for your four sorting functions for $n = 500, 1000, 2000, 3000, 4000, 5000$. To time an event that is smaller than or near the clock accuracy. repeat it many times and divide the overall time by the number of repetitions.You should obtain times that are accurate to within 1%.

1.2 WHAT TO TURN IN

1. Figure out the clock accuracy.
2. The number of random permutations tried for Heap Sort, the worst-case data for Merge Sort and how to generate it.
3. A table of times for the above values of n, the times for the narrowed ranges, the graph, and a table of times for the composite function.
4. A complete listing of the program (this includes the sorting functions and the main program for timing and test-data generation.

2 Basic Idea

2.1 Program Structure

Looking at the big picture, I implemented a total of five classes. They are `SortUnit` and classes containing five sorting algorithms and data generation functions, among which the five classes containing sorting algorithms inherit from `SortUnit`.

In `SortUnit`, use the C++ programming technique CRTP to realize static polymorphism, and only use the function `do_experiment` to complete the experiment for a certain sorting algorithm.

2.2 Data Generation

2.2.1 Insertion Sort

In this case, just use sequence $n-1, n, \ldots, 0$. On the other hand, it's also feasible to generate random data and then sort it reversely. The code is shown below, note that I used the range library of C++20.

```
1 static vector<int> generate(int n) {
2     return views::iota(0, n) | views::reverse | ranges::to<vector>();
3 }
```

2.2.2 Merge Sort

Worst-case data for merge sort can be obtained by working backward. Begin with the last merge your function will perform and make this work hardest. Then look at the second-to-last merge, and so on. Use this logic to obtain a program that will generate worst-case data for Merge Sort for each of the above values of n.

In this code, I even used the latest feature of c++23 (explicit this parameter), but I used conditional compilation to ensure that my program can run successfully under the lower version of the language standard.

```
1 static vector<int> generate(int n) {
2     auto arr = views::iota(0, n) | ranges::to<vector>();
3 #ifdef __cpp_explicit_this_parameter
4     auto separate = [](this auto self,
5 #else
```

```
    std::function<void(vector<int>&)> separate = [](
#endif
                          vector<int>& nums) -> void {
        if (nums.size() == 2)
            swap(nums[0], nums[1]);
        else if (nums.size() > 2) {
            vector<int> left  = nums |
                views::stride(2) | ranges::to<vector>();
            vector<int> right = nums | views::drop(1) |
                views::stride(2) | ranges::to<vector>();
#ifdef __cpp_explicit_this_parameter
            self(left);
            self(right);
#else
            separate(left);
            separate(right);
#endif
            ranges::copy(left, nums.begin());
            ranges::copy(right, nums.begin() + left.size());
        }
    };
    separate(arr);
    return arr;
}
```

2.2.3 Heap Sort

It's difficult to generate the worst case for heap sort. This is because, for heap sort, the best and worst cases are $\Theta(nlogn)$ - assuming all elements are distinct - which only means that asymptotically there is no difference between the two, although they can differ by one constant factor.

Therefore, I generate 10 integer permutations randomly by using `ranges::shuffle` instead of `Permute` given in title. Then, pick the most time-consuming permutation to approximate to the worst-case time.

```
static vector<int> gen_most_time_consuming(int n) {
    array<vector<int>, 10>                      data;
    array<chrono::duration<double, micro>, 10> time{};
    data.fill(views::iota(0, n) | ranges::to<vector>());
    for (auto i : views::iota(0, static_cast<int>(data.size())))
        time[i] = cal_time(1, n, [&](const auto) {
            ranges::shuffle(data[i], mt19937{ random_device{}() });
            return data[i];
        });
    return data[ranges::distance(
        time.begin(), ranges::max_element(time))];
}
```

2.2.4 Quick Sort

In quick sort, consider choosing the absolute minimum of each array as a pivot. Then (roughly) n compares are done at the top layer, then (roughly) $n-1$ in the next layer, then (roughly) $n-2$ in the next, etc. The sum $1+2+3+\cdots+n$ is $\Theta(n^2)$, hence the worst case.

It is easy to generate a suitable worst-case permutation for ordinary quicksort, i.e. choose the smallest element as the pivot in increasing order. However, by choosing the median as the pivot, this prevents the worst-case scenario to some extent. If we want to generate worst-case permutation, we have to do as we did in previous section by invoking `gen_most_time_consuming`.

2.3 Algorithm Design

The title requires an algorithm that combines four sorting algorithms to achieve the best worst-case performance. But I went a step further and designed an algorithm that has the best performance at any time. It dynamically selects the appropriate sorting algorithm according to the data size and data characteristics. I will introduce it below.

This is a sorting algorithm that combines quick sort, heap sort, and insertion sort. When the amount of data is large, quick sorting is used, and the determination of pivot adopts the method of taking the middle of three numbers. When the data is divided to a threshold, in order to prevent deterioration, heap sorting is used. Finally, when the sequence is almost sorted, use insertion sort.

The pseudo code is as follows:

Algorithm 1 OPTIMAL-SORT

```
function OptimalSort(first, last, ideal)
    while True do
        if last - first ≤ THRESHOLD then
            InsertionSort(first, last) return
        if ideal ≤ 0 then
            HeapSort(first, last) return
        mid = Partition_by_median(first, last)                ▷ from quick sort
        ideal = (ideal >> 1) + (ideal >> 2)                   ▷ allow 1.5log2(N) divisions
        if mid − first < last − mid then
            OPTIMALSORT(first, mid, ideal)
            first = mid ‘
        else
            OPTIMALSORT(mid, last, ideal)
            last = mid
```

`ideal` is a dynamically changing threshold used to detect whether there are too many intervals in quick sorting. If too much ($ideal \leq 0$), call heapsort. Initially, it is set to the size of the range to be sorted.

THRESHOLD is the threshold used to judge the size of the current interval to be sorted. When the data to be sorted is small, the performance of insertion sort is usually better, so use insertion sort when the data is smaller than this threshold. Usually, it is set to 32.

3 Procedure

I calculated the worst case running time for each algorithm. Run when $n = 500, 1000, 2000, 3000, 4000, 5000$, and repeat 10 times for each n to calculate the average time.

```cpp
void do_experiment(ostream& os) {
    constexpr int n[] = { 500, 1000, 2000, 3000, 4000, 5000 };
    os << Derived::category << ",";
    for (auto cnt : n)
        os << cal_time(EXPERIMENT_N, cnt) << ",";
    os << endl;
}

template <class Gen>
auto cal_time(int repeat_number, int n, Gen gen) {
    chrono::duration<double, micro> sum{ 0 };
    vector<int>                     data = gen(n);
    for (auto _ : views::iota(int{ 0 }, repeat_number)) {
        auto start = chrono::high_resolution_clock::now();
        Derived::sort(data);
        auto end = chrono::high_resolution_clock::now();
        sum += chrono::duration_cast<decltype(sum)>(end - start);
```

```
18        }
19        return sum * 1.0 / repeat_number;
20    }
```

4 Result and Analysis

I record the test data of each sorting algorithm as table 1.

	500	1000	2000	3000	4000	5000
insertion sort	164.92	646.88	2590.61	5835.37	10431.2	16480.5
merge sort	549.21	1099.09	2378.28	3743.15	5018.81	6705.92
heap sort	228.48	505.27	1125.84	1781.15	2776.46	3378.72
quick sort	71.16	98.66	214.44	341.44	473.55	567.97
optimal sort	16.58	38.4	87.64	143.94	203.92	353.77

Table 1: Performance table of each sorting algorithm. The time unit is μs

I also plot the data in the figure 2. It can be seen that the worst-case growth curve of insertion sort is indeed $O(n^2)$. However, since the data scale is still not large enough, the growth trend of several other algorithms is not obvious enough. Nevertheless, we can get the final result through mathematical analysis.

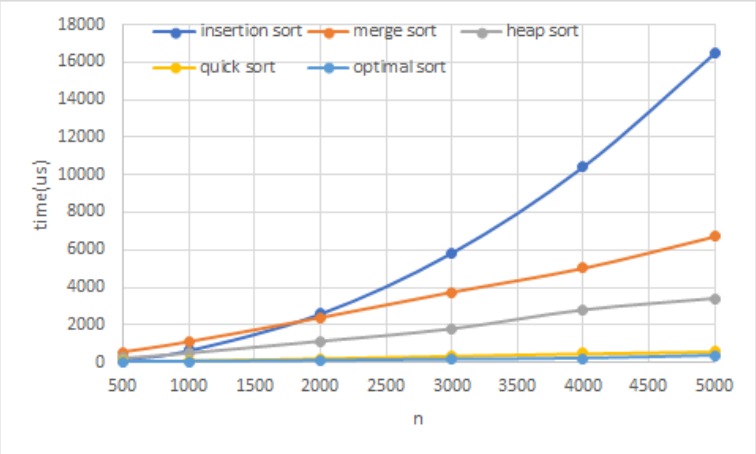


Figure 2: Performance figure of each sorting algorithm.