CSC 212: Data Structures and Abstractions

14: Sorting

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

THINK BIG WE DO℠

Sorting

- Given a sequence of *n* elements that can be compared according to a total order relation
  - ✓ we want to rearrange them in monotonic order (non-decreasing or non-increasing)
- Formally, the output of any sorting algorithm must satisfy two conditions:
  - ✓ the output is in monotonic order (each element is no smaller/ larger than the previous element, according to the required order)
  - ✓ the output is a permutation (a reordering that retains all of the original elements) of the input

`central problem in computer science`

Insertion sort

- Algorithm (non-decreasing order)
  - ✓ start at index 1, loop through the array
  - ✓ for each element
    - compare with the element to its left
    - if smaller, swap them and move left
    - repeat until element is not smaller or you reach the start

Time complexity depends on the input

- Worst-case: $\Theta(n^2)$
  - input is reverse sorted
- Best-case? $\Theta(n)$
  - input is already or almost sorted
- Average-case? $\Theta(n^2)$
  - expect every element to move $O(n/2)$ times on average

```
void insertion_sort(std::vector<int>& A) {
    for (size_t i = 1 ; i < A.size() ; ++i) {
        for (size_t j = i; j > 0 ; --j) {
            if (A[j] < A[j-1]) {
                std::swap(A[j], A[j-1]);
            } else {
                break;
            }
        }
    }
}
```


Merge sort

Merge sort

- **Divide** the array into two halves
  - ✓ calculate the midpoint to split the array
- **Conquer** each half recursively
  - ✓ call merge sort on each half (solve 2 smaller problems)
- **Combine** the solutions
  - ✓ after both recursive calls finish, **merge** the two sorted halves into one sorted array

Divide and Conquer Methods
A problem-solving approach that breaks a problem into smaller subproblems, solves them independently, and then combines their solutions

Examples: binary search, merge sort, quick sort

Merge sort algorithm

```
if (hi <= lo) return

mid = lo + (hi - lo) / 2

mergesort(A, lo, mid)
mergesort(A, mid+1, hi)

merge(A, lo, mid, hi)
```

```
void merge_sort(std::vector<int>& A, size_t lo, size_t hi) {
    if (hi <= lo) return;
    size_t mid = lo + ((hi-lo) / 2);
    merge_sort(A, lo, mid);
    merge_sort(A, mid+1, hi);
    merge(A, lo, mid, hi);
}
```


Merge sort

0	1	2	3	4	5	6	7
4	2	1	10	6	20	7	30

Merging two sorted arrays

lo			mid				hi
1	2	4	10	6	7	20	30

i → j →

k →

A temporary array is necessary to guarantee a linear time operation

Show me the code

```
void merge(std::vector<int>& A, size_t lo, size_t mid, size_t hi) {
    std::vector<int> B(hi - lo + 1);
    size_t i = lo, j = mid + 1, k = 0;

    while (i <= mid && j <= hi) {
        if (A[i] <= A[j]) {
            B[k++] = A[i++];
        } else {
            B[k++] = A[j++];
        }
    }
    while (i <= mid) B[k++] = A[i++];
    while (j <= hi) B[k++] = A[j++];

    for (k = 0 ; k < B.size() ; ++k) {
        A[lo + k] = B[k];
    }
}
```


Draw the recursion call tree

- What is the complexity?
  - ✓ best case?
  - ✓ worst case?
  - ✓ average case?

```
void merge_sort(std::vector<int>& A, size_t lo, size_t hi) {
    if (hi <= lo) return;
    size_t mid = lo + ((hi-lo) / 2);
    merge_sort(A, lo, mid);
    merge_sort(A, mid+1, hi);
    merge(A, lo, mid, hi);
}
```


Analysis of merge sort

$$T(0) = 0$$
$$T(1) = 0$$
$$T(n) = 2T(n/2) + \Theta(n)$$


```
void merge_sort(std::vector<int>& A, size_t lo, size_t hi) {
    if (hi <= lo) return;
    size_t mid = lo + ((hi-lo) / 2);
    merge_sort(A, lo, mid);
    merge_sort(A, mid+1, hi);
    merge(A, lo, mid, hi);
}
```


count the number of comparisons

Final comments

- Major disadvantage
  - ✓ a sorting algorithm is in-place if it uses O(log n) extra memory
  - ✓ merge sort is not **in-place**
- Improvements
  - ✓ use insertion sort for small arrays
  - ✓ stop recursion if subarray already sorted