CS314H CSB Lecture 13: Sorting

Sorting

2. Sorting & the original order
   - _________ sort: maintains the original order of “equal” items.
   - _________ sort: does not maintain the order.

3. Sorting & space usage
   - ____________ sort: does not require additional memory just use original space for sorting.
   - ____________ sort: use extra space that's directly related to the size of the problem.
4. When is stable sort useful?

5. How to prove correctness of a sorting algorithm?
   Use _______________________- A loop property that is true at the start or at the end of each iteration.

6. Can you find a loop invariant for below code?

```
int calcSum(int array[], int n){
    int sum = 0;
    for(int i = 0 ; i < n ; i++){
        sum += array[i];
    }
    return sum;
}
```

After 1^{st} iteration, sum contains what?
After 2^{nd} iteration, sum contains what?
…
Loop invariant: After k^{th} iteration, sum contains ______________________________.

Selection Sort

7. For each iteration you are allowed to do only one swap.
8. Find max approach: Find max among index ____thru____ and swap the max with element at array[____], where k goes from _____ to ________.

9. Find min approach: Find min among index ____thru____ and swap the max with element at array[____], where k goes from _____ to ________.

10. Code (select min and swap)

```
void selectionSort(vector<int>& list) {
    for(int i = 0 ; i < list.size() - 1 ; i++) {
        //Before iteration with index i, list[0:i-1] is sorted
        //find min index from unsorted list[i:n-1]
        int min = i;
        for(int j = i + 1 ; j < list.size() ; j++) {
            if(list[min] > list[j]) {
                min = j;
            }
        }
        //send the min value to the start of unsorted list
        swap(list[i], list[min]);
        //After iteration with index i, list[0:i] is sorted
    }
}
```

- Is it stable?
- Is it in-place? What is the space complexity?
- Loop invariant?
- Time complexity?

11. Best case input vs worst case input?
    - Best case (no swap needed):
    - Worst case (always need to swap):
    - Time complexity for each?

Bubble sort

12. Find max and place it to the rightful max's place by swapping adjacent items. (multiple swaps are allowed).
13. Code

```
void bubbleSort(vector<int>& list) {
    for(int i = (int) list.size() - 1 ; i > 0 ; i--) {
        //Before iteration with index i, list[i+1:n-1] is sorted
        //bubble up max of unsorted list[0:i] to list[i]
        for(int j = 0 ; j < i ; j++) {
            if(list[j] > list[j + 1]) {
                swap(list[j], list[j + 1]);
            }
        }
        //After iteration with index i, list[i:n-1] is sorted
    }
}
```

- Is it in-place?
- Is it stable?
- Loop invariant?
- Time complexity

14. Best case vs worst case input and time complexity for each case?
    - Best case (no swap needed at all):
    - Worst case (always need to bubble up all the way):

Insertion sort

15. Shifting is allowed (do not need to rely on swapping). For each item at index k in the array find a right hole j between 0..k inclusive and insert to the hole by shifting later items (items at index j+1 … k-1) to the right.

```
void insertionSort(vector<int>& list) {
    for(int i = 1 ; i < list.size() ; i++) {
        //Before iteration with index i, list[0:i-1] is sorted
        //find a hole to place list[i] from sorted list[0:i-1]
        int val = list[i];
        int hole = i;
        while (hole > 0 && val < list[hole - 1]) {
            list[hole] = list[hole - 1]; //shift right
            hole--;
        }
        list[hole] = val;
        //After iteration with index i, list[0:i] is sorted
    }
}
```

- Stable?
- In place?
- Loop invariant?
- Time complexity

16. Best case vs worst case input

Mergesort

17. Mergesort is a _______-and-________ algorithm: Divide a large problem into smaller problems andsolve the smaller problem to solve the large problem.
    - Divide the list into two roughly equal halves.
    - Sort the _____ half.
    - Sort the _____ half.
    - ________ the two sorted halves into one sorted list.

18. Mergesort is a ____________ algorithm: an algorithm that solves a problem by breaking it down into smaller instances of the same problem. Consists of
    - _________ case: The simplest instance of the problem that can be solved directly without recursion.
    - ___________ case: The part where the algorithm calls itself to solve a smaller version of the original problem.

19. Let’s first implement merge. What should be the if condition?

```
//a version of merge used for mergeSort2
//merges two separate halves (left and right) into list
//pre-condition: 1) left and right is sorted
//               2) size of list is equal to size of left + size of right
void merge2(vector<int>& list, const vector<int> left, const vector<int> right){
    int i1 = 0; //left index
    int i2 = 0; //right index
    for(int i = 0 ; i < list.size(); i++){
        if(i1 < left.size() &&
           (i2 >= right.size() || left[i1] <= right[i2])) {
            list[i] = left[i1++]; //take from left
        } else {
            list[i] = right[i2++]; //take from right
        }
    }
}
```

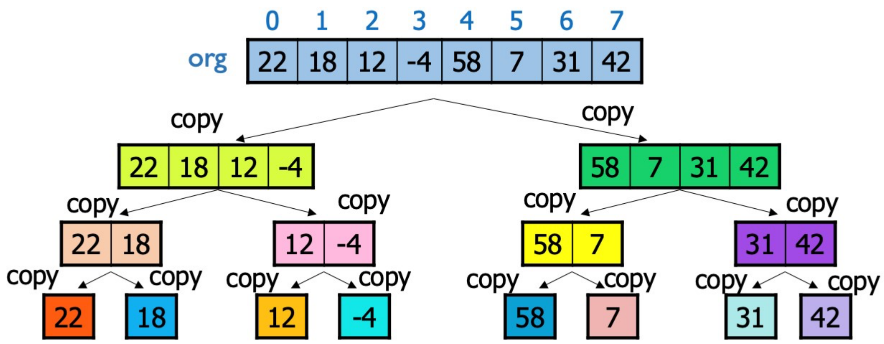
- Is it stable?
- What it the time complexity of just merge?

20. Complete below code:

```
//sub-optimal merge sort
void mergeSort2(vector<int>& list){
    if(list.size() <= 1) return; //base case

    // copy list into two halves (left and right)
    vector<int> left(list.size()/2);
    vector<int> right(list.size() - left.size());
    std::copy(list.begin(), list.begin() + left.size() ,left.begin());
    std::copy(list.begin() + left.size(), list.end(), right.begin());

    // sort the two halves
    mergeSort2(left);
    mergeSort2(right);

    // merge the sorted havles into list
    merge2(list, left, right);
}
```

21. What is the space complexity?

• Informal analysis:



At each recursion level, we see exactly ________ cells.
How many levels are there? n is reduced to 1 by dividing into half each time. __________ levels.

What is the space complexity (the max space used at the deepest level of recursion)?

22. Formal analysis using the recurrence relation:
    Let S(n) be the space needed for mergeSort with input size n. Write the recurrence for S(n):

23. Space complexity does not look good. How to save space?
    Where/which part of the code actually needs this space? Circle one from below.
    - Divide the list into two roughly equal halves.
    - Sort the left half.
    - Sort the right half.
    - Merge the two sorted halves into one sorted list.

24. One way to improve this code is to move the creation of new array inside the merge function.
    - Why the simply moving instantiation and array copy code to `merge()` saves space?

```cpp
// Function to merge two sorted subarrays into one sorted array
void merge3(vector<int>& array, int left, int mid, int right) {
    // Calculate the sizes of the two subarrays
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays to hold the elements
    vector<int> L(n1);
    vector<int> R(n2);

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = array[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = array[mid + 1 + j];

    // Merge the two temporary arrays back into the original array
    // Code ommited
      ...
    }
}

// Function to implement merge sort
void mergeSort3(vector<int>& array, int left, int right) {
    if (left >= right) return;
    // Find the middle point
    int mid = left + (right - left) / 2;

    // Recursively sort the two halves
    mergeSort3(array, left, mid);
    mergeSort3(array, mid + 1, right);

    // Merge the sorted halves
    merge3(array, left, mid, right);
}
```

Draw space usage diagram and see if space complexity is indeed O(n)

Compare memory usage with 22.

25. Final approach! We do not like to create/de-create objects each time merge is called. We want to just start with n extra space to begin with.

Draw the space usage diagram.

26. Complete the code (mergeSort final version).

```
void mergeSort(vector<int>& list){
    vector<int> copy(list);  // copy of array
    //copy is the source and list is the output
    mergeSort(copy, list, 0, (int) list.size() - 1);
}
```

```
//sort list[start:end] and save to result[start:end]
//start and end is inclusive
void mergeSort(vector<int>& list, vector<int>& result, int start, int end) {
    if (end - start <= 0) return; // base case (size 1 or 0 is already sorted)
    int mid = (start + end) / 2;

    // sort the two halves
    mergeSort(result, list, start, mid); //sort left and save to list
    mergeSort(result, list, mid + 1, end); //sort right and save to list

    // merge the sorted halves into a sorted whole
    merge(list, result, start, mid, mid + 1, end); //save the merged to result
}
```

27. Time complexity. Write down the recurrence for merge sort.

28. General strategy: How to solve recurrence?

- By ____________: Directly expand and substitute terms recursively. Ex) $T(1) = 0$

  $T(n) = 1 + T(n-1)$ //substitute $T(n-1)$

  $= 1 +$ //substitute $T(n-2)$

  $= 1 +$ //substitute $T(n-3)$

  …

  $= 1 +$ //substitute $T(1)$
- By ____________: Manipulate terms to achieve cancellation. Add all the left-hand side up and right hand-side up and cancel items

  Ex)

  $T(n) = 1 + T(n-1)$

  $T(n-1) = 1 + T(n-2)$

  After cancelling out, what is left?

  $T(n) =$

29. Solve recurrence of merge sort.

$$T(n) = \begin{cases} \Theta(1) & \text{if } (n = 0) \\ \Theta(1) & \text{if } (n = 1) \\ \Theta(n) + 2\,T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

- By substitution? Tricky (this strategy typically works better with linear recurrence)

- By telescoping! (Hint: re-write the recurrence by dividing by n)

- By art (my favorite ☺): Formally this is called _______________.

Quick sort

30. Psuedo code: what is wrong with below code?

```
qsort (List S) {      if (|S| <= 1) return S;// Note that

S could be empty      v = element of S;// Choose pivot

    // Using set notation for lists
    List L = { x in  S - {v} | x <= v }
List R = { x in  S - {v} | x >= v }

    return (qsort(L) + {v} + qsort (R));// + is list concatenation }
```

Assume x can go either L or R (not both!) Why not make it deterministic?

```
    // Using set notation for lists
    List L = { x in  S - {v} | x <= v }
    List R = { x in  S - {v} | x > v }
```

Partitioning is the key to the performance of this algorithm.
31. What is a bad pivot?
Write a recurrence for using a bad pivot for each iteration.
Bad pivot analysis by telescoping (Worst case analysis).

32. What is a good pivot?
Write a recurrence for using a good pivot for each iteration

33. How to choose a pivot?

Which one is worse 1) random vs 2) median of any 3 elements?