DSA

WEEK 2

DISCUSSION SESSION

Pseudocode analysis: iterative/recursive

- Q: How to translate from simple operations (steps) to asymptotical?
- Q: How to analyze recursive time complexity?
- Q: Iterative or recursive manner: which of them is the most efficient?

Insertion sort

Iterative or recursive manner, which of them is the most efficient?

```
function INSERTIONSORT_I(A)
    for n from 2 to A.length
        x = A[n], j=n
        while j >= 2 and A[j-1] > x do
                A[j]=A[j-1], j=j-1
        end
        A[j]=x
    end
end
```

```
function INSERTIONSORT_R(A,n)
    if n > 1
        INSERTIONSORT_R(A, n-1)
        x = A[n], j=n-1
        while j >= 1 and A[j] > x do
                A[j+1]=A[j], j=j-1
        end
        A[j+1]=x
    end
end

INSERTIONSORT_R(A, A.length)
```

https://en.wikipedia.org/wiki/Insertion_sort

Insertion sort

- https://www.geeksforgeeks.org/dsa/insertion-sort-algorithm/
- https://www.geeksforgeeks.org/dsa/recursive-insertion-sort/

iterative.cpp U

insertion_sort > iterative.cpp > ...

```cpp
// C++ program for implementation of Insertion Sort
#include <iostream>
using namespace std;

/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are greater than key,
           to one position ahead of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver method
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
```

recursive.cpp U

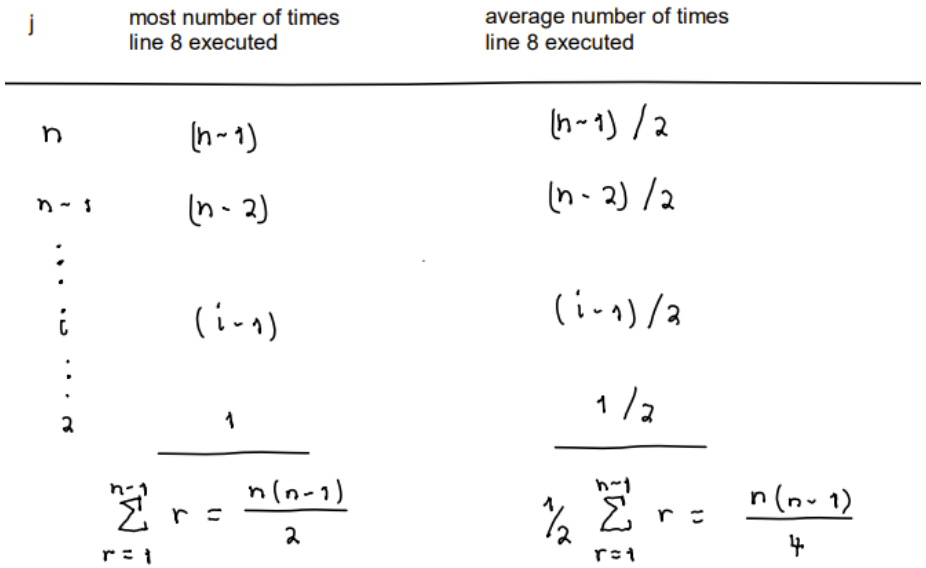
insertion_sort > recursive.cpp > ...

```cpp
// Recursive C++ program for insertion sort
#include <iostream>
using namespace std;

// Recursive function to sort an array using insertion sort
void insertionSortRecursive(int arr[], int n)
{
    // Base case
    if (n <= 1)
        return;

    // Sort first n-1 elements
    insertionSortRecursive( arr, n-1 );

    // Insert last element at its correct position in sorted array
    int last = arr[n-1];
    int j = n-2;

    /* Move elements of arr[0..i-1], that are greater than key,
    to one position ahead of their current position */
    while (j >= 0 && arr[j] > last)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    for (int i=0; i < n; i++)
        cout << arr[i] <<" ";
}

/* Driver program to test insertion sort */
int main()
{
    int arr[] = {12, 11, 13, 5, 6};
```

Complexity of insertion sort

- These are the operations that are done by the Insertion Sort algorithm for the first elements:
  - The 1st value is already in the correct position.
  - The 2nd value must be compared and moved past the 1st value.
  - The 3rd value must be compared and moved past two values.
  - The 3rd value must be compared and moved past three values.
  - And so on..
  - If we continue this pattern, we get the total number of operations for n values:
  - 1+2+3+...+(n−1)
- This is a well-known series in mathematics that can be written like this: $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$
- For very large n, the $\frac{n^2}{2}$ term dominates, so we can simplify complexity for the Insertion Sort algorithm: $O(\frac{n^2}{2})$ =$O(n^2)$

https://www.w3schools.com/dsa/dsa_timecomplexity_insertionsort.php



Q: The part that remains unclear is the analysis for the "average case". I don't fully understand the assumption that we only check half of the sorted part on average.

Binary search

Iterative or recursive manner, which of them is the most efficient?

```
function BINARY_SEARCH_I(A, T)
    L = 1
    R = A.length
    while L ≤ R do
        mid = ⌊(L+R)/2⌋
        if A[mid] < T then
            L = mid + 1
        else if A[mid] > T then
            R = mid - 1
        else
            return mid
    end
    return unsuccessful
End
```

```
function BINARY_SEARCH_R(A, L, R, T)
    if L == R then
        if A[L] == T then
            return L
        else
            return unsuccessful
        end
    else
        mid = ⌊(L+R)/2⌋
        if T <= A[mid] then
            return BINARY_SEARCH_R(A,L,mid,T)
        else
            return BINARY_SEARCH_R(A,mid+1,R,T)
        end
end
```

https://en.wikipedia.org/wiki/Binary_search

Binary search

- https://www.geeksforgeeks.org/dsa/binary-search/

binary_search > iterative.cpp > main()

```cpp
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(vector<int> &arr, int x) {
    int low = 0;
    int high = arr.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;

        // Check if x is present at mid
        if (arr[mid] == x)
            return mid;

        // If x greater, ignore left half
        if (arr[mid] < x)
            low = mid + 1;

        // If x is smaller, ignore right half
        else
            high = mid - 1;
    }

    // If we reach here, then element was not present
    return -1;
}

int main() {
    vector<int> arr = { 2, 3, 4, 10, 40 };
    int x = 10;
    int result = binarySearch(arr, x);
    if(result == -1) cout << "Element is not present in array";
    else cout << "Element is present at index " << result;
    return 0;
}
```

binary_search > recursive.cpp > ...

```cpp
#include <iostream>
#include <vector>
using namespace std;

// A recursive binary search function. It returns
// location of x in given array arr[low..high] is present,
// otherwise -1
int binarySearch(vector<int> &arr, int low, int high, int x) {
    if (high >= low) {
        int mid = low + (high - low) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, low, mid - 1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid + 1, high, x);
    }
  return -1;
}

int main() {
    vector<int> arr = { 2, 3, 4, 10, 40 };
    int query = 10;
    int n = arr.size();
    int result = binarySearch(arr, 0, n - 1, query);
    if (result == -1) cout << "Element is not present in array";
    else cout << "Element is present at index " << result;
    return 0;
}
```

Complexity of binary search

- The **worst-case scenario** is if the search area must be cut in half over and over until the search area is just one value. When this happens, it does not affect the time complexity if the target value is found or not.
- Let's consider array lengths that are powers of 2, like 2, 4, 8, 16, 32, 64 and so on.
- How many times must 2 be cut in half until we are looking at just one value? It is just one time, right?
- How about 8? We must cut an array of 8 values in half 3 times to arrive at just one value.
- An array of 32 values must be cut in half 5 times.
- We can see that $2=2^1$, $8=2^3$ and $32=2^5$. So, the number of times we must cut an array to arrive at just one element can be found in the power with base 2. Another way to look at it is to ask "how many times must I multiply 2 with itself to arrive at this number?". Mathematically we can use the base-2 logarithm, so that we can find out that an array of length n can be split in half $\log_2(n)$ times.
- Binary search is: $O(log_2 n)$
- https://www.w3schools.com/dsa/dsa_timecomplexity_binarysearch.php

Recursion

- Q: “Stack" and “base case” in recursive procedures?
- Q: Why it just does not stop there like "return 1, end“?
- Tail recursion
  - tail call = the last action of the procedure
  - tail recursion = last action is to call itself
  - can be optimized easily e.g. implemented without adding a new stack frame to the call stack

https://en.wikipedia.org/wiki/Tail_call

Benefits of iterative vs. recursive

- What are pros and cons?
- Q: When to use recursion instead of a loop (iteration)

Different sorting algorithms and all the types of problems they can be applied to

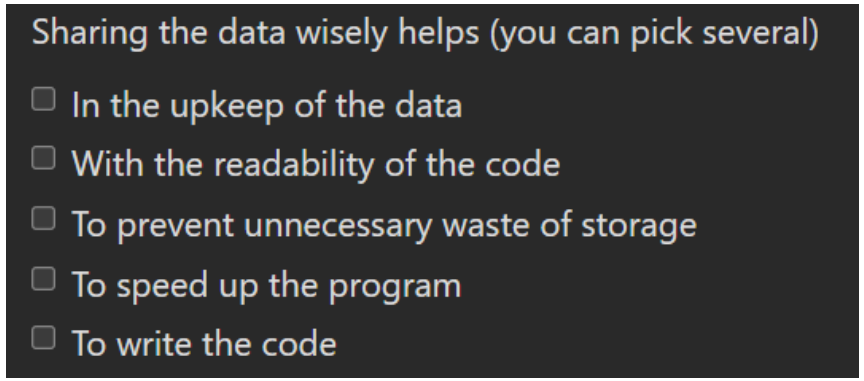
- Q: Is it's feasible to mix some of them together to create a more efficient algorithm
  - https://en.cppreference.com/w/cpp/algorithm/sort.html

  Introsort or **introspective sort** is a hybrid sorting algorithm that provides both fast average performance and (asymptotically) optimal worst-case performance. It begins with quicksort, it switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted and it switches to insertion sort when the number of elements is below some threshold. This combines the good parts of the three algorithms, with practical performance comparable to quicksort on typical data sets and worst-case O(n log n) runtime due to the heap sort. Since the three algorithms it uses are comparison sorts, it is also a comparison sort

  https://en.wikipedia.org/wiki/Introsort

Sharing data

- Q: It's not totally clear how we decide when to copy data versus just referring to it indirectly in a real program.
- Q: Does sharing data using indices of the master data structure prevent us from getting into reference invalidation, as the case 1? Sure, if we have the objects in a map data structure, we can refer to the ID number, but using the indices of the vector element position can definitely be modified and lead to reference invalidation?
- Q: To me, I would have argued that all the selectable options would have benefitted with sharing data with data structures (as opposed to not doing that).



How to solve insertion sort

- First click the current element to store it as temp, then repeatedly compare it with elements to its left. For each larger element found, click it and then click its right neighbor to shift it right. Finally, click the empty position to place temp. The key is performing every shift step without skipping any.

  https://plus.tuni.fi/COMP.CS.300/fall-2025/week1/content/