Asgn 5 DESIGN.pdf

1.1 Bubble Sort.

2 1 4 3

2 1 3 4

Smallest number goes front.

Pre lab part 1

(1)

8 22 7 9 31 5 13

5 8 22 7 9 31 13

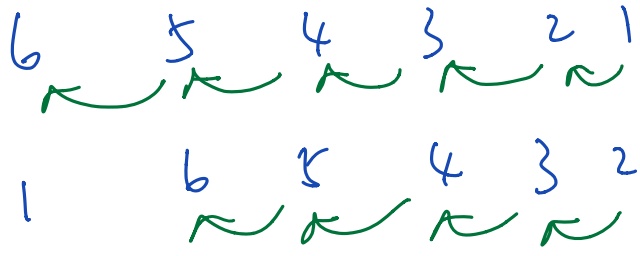
5 7 8 22 9 13 31

5 7 8 9 22 13 31

12 swaps

(2) $\Sigma (n-i)$ comparisons.

because $(n-1)+(n-2)+\dots$



Pre-lab part 2

1. Shell sort time complexity depends on the gap size because the gap size determines how many comparisons the program need to execute. Depends on the size of the array, we can adjust the size sequence accordingly. If we reduce the size of the gap.

2. Reduce the number of nested loops.

Pre-lab Part 3

1. Quicksort with have time complexity of $O(n^2)$ only if the array is sorted. So the partition function keep getting $(n-1)$ pivot and not able to divide the array into two.

Most people use Quicksort when the array is not sorted, so $O(n^2)$ is not a usual occanrance.

Pre lab part 4

Since array insertion take $O(n)$. because you basicaly shifting all the element, so it won't be that efficent compare to other sorting algorithm.

Pre lab part 5

1. Declare variable in Array.h, move & compare, these 2 variables keep track on their's own array if modified.

Design

Files : bubble.h . c

Shell. h . c

quick. h . c

binary. h . c

Sorting. c — main()

Main

option = "Absqip:r:n:"

- A // employ all algorithms
- b // bubble sort
- s // shell sort
- q // Quick Sort
- i // Binary Insertion Sort
- p n // print first n element of array
  n = 100 default
- r s // random seed
  r = 8222022
- n c // set array size to c
  // c = 100

Switch (option) {

-option // will declare boolean value.

bool = true if option is entered.
it set the boolean value.

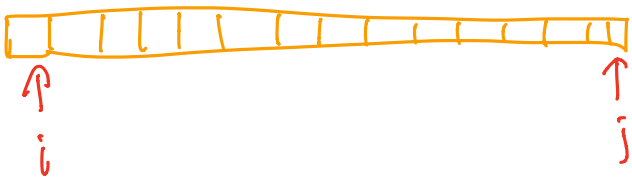
according to the boolean value.
will call the functions after

}

switch ( ).

Bubble Sort

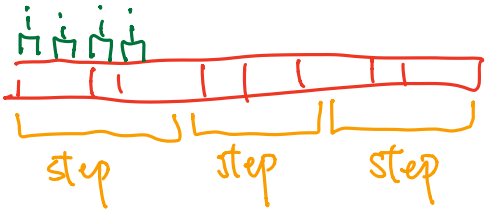
```
void Bubble_Sort (int * arr) {
// void function, all swap in function
    for (arr : i) { // for i in arr
```



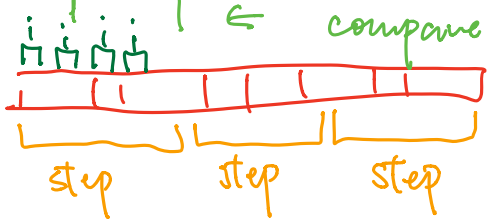

```
    while j>i { // while j still on the right
    if arr[j] < arr[j-1]
        swap [j, j-1]      // swap j & j-1 if they
    j--;                       not in order
    }
// the left most element should be
   smallest at this point.
```

Shell Sort

```
void Shell_Sort (int* arr) {// take arr as parameter.
    for step in gap(arr)  // array.
                          // loop between gap
```




```
        for i in (step to length) // starting from gap.
            for j in (i to step)
                if j < j - step
```

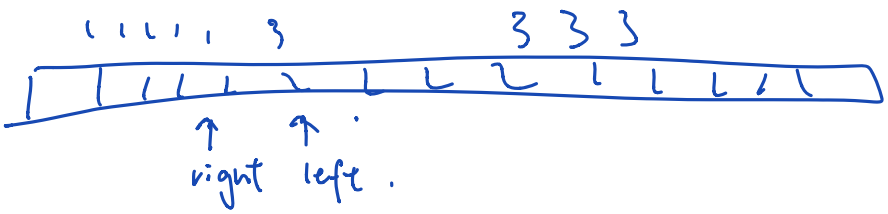


compare this two.

```
                    swap().
                    // compare if the number that is a gap away is in the right position.
                    if not, swap them
```

Quick_sort

```
int partition (arr, left, right)
                                    // return an index that
pivot = arr[left]                       has been properly sorted.

lo = left + 1   // declare low and high,
hi = right          we are arranging value
                    between low & high

while {
    while lo <= hi & arr[hi] >= pivot {
        hi-- }  // stop until we find something
                    on the right is smaller
                    than pivot.

    while lo <= hi and arr[lo] <= pivot {
        lo ++ }  // stop until left > pivot.


if lo <= hi   // if the pointer still in right place
    swap [left, right] // swap, since left & right
else                    stop at the place that
    break --> get out the loop.     they should be swap.

---

swap (left . hi ). // swap pivot and high, so pivot
    return hi           get to the right place.
```




```
void Quick_sort (arr, left, right)
    if left < right
        index = partition ().  // will sort array in
                                  two, index is the final
                                  position.
    Quick sort (index+1, right)
    Quicksort (left, index-1)  // call sort () twice
                                  to sort those seperate
                                  array.
```

```
void Binary_Insertion_Sort (arr)

for i in (1 to length) {

    value = arr[i]  // current value

    left = 0

    right = i

    while left < right {

        mid = left + (right - left)/2  // mid point btw left & right
```


```
        if value >= arr[mid] }  // left / right meet tgt eventually.

            left = mid + 1

        else

            right = mid.
```

to change the if right position if right in the right position, left moves.

```
    for j in (i to left)

        swap(arr[j-1], arr[j])
```

// insert i to the approximate position.

For loop because we need to shift every other element —

Binary

moves ✓ Compares ✓

Quick

moves ✗ Compares ✗

Shell

moves ✓ Compares ✓

Bubble

moves ✓ Compares ✗