# ASSIGNMENT 5 - SORTING

## BUBBLE SORT

- SECOND < FIRST : SWAP
  - LARGEST SORTED → n-1 NEXT
- DONE WHEN NO PAIRS ARE OUT OF ORDER

- WORST CASE $O(n^2)$

PSUEDOCODE: (FROM DOC)

- FOR i = 0, i < len(arr)-1 :
  - j = len(arr)-1
  - while j > i : comparison
    - check if arr[left] is < arr[right]
      - if so swap 3 moves
    - decrement j

## PRE LAB PART 1

1) HOW MANY ROUNDS OF SWAPPING DO YOU THINK YOU WILL NEED TO SORT THE NUMBERS 8, 22, 7, 9, 31, 5, 13 IN ASCENDING ORDER WITH BUBBLE SORT?

BECAUSE THE LENGTH OF THE ARRAY IS 7, YO WILL NEED TO DO 6 ROUNDS OF SWAPPING AT MOST BECAUSE IT IS ONE LESS THAN THE LENGTH. IN THIS SCENARIO THE SMALLEST # '5' IS THE SECOND TO LAST ELEMENT, MEANING IT WILL ONLY MOVE 1 SPOT LEFT EVERY ITERATION, AND WILL TAKE THE MAX NUMBER OF ROUNDS.

2) HOW MANY COMPARISONS CAN WE EXPECT TO SEE IN THE WORSE CASE SCENARIO FOR BUBBLE SORT?

BUBBLE SORT WILL DO THE SAME # OF COMPARISONS IN ITS WORST CASE + BEST CASE — $\sum_{i=1}^{n-1} i$

## SHELL SORT

- SORTS ELEMENTS FAR APART WITH A SHRINKING GAP

- IF $n \leq 2$ : $n = 1$ ELSE $n = 5 * n // 11$

**PSUEDO CODE:** (FROM CAB MANUAL)

GAP:
- while n > 1:
  - n = 1 if n <= 2 ELSE SA n// 1|
  - USE THIS VALUE (YIELD) OR ADD TO ARRAY

SHELL_SORT:

- FOR STEP IN GAP:
  - FOR i = STEP, i < len(arr):
    - FOR j in range(i, step - 1, -step):
      - if arr[j] < arr[j-step]: comparison
        - swap arr[j], arr[j-step] 3 moves

## PRE LAB 2

1) SHELL TIME COMPLEXITY DEPENDS ON GAP SIZE. WHY? HOW CAN YOU IMPROVE THE TIME COMPLEXITY BY CHANGING GAP SIZE?

THE GAP DETERMINES THE DISTANCE BETWEEN ELEMENTS TO COMPARE. GOING THROUGH TOO FEW GAPS SLOWS THE TIME, BUT TOO MANY CREATES AN OVERHEAD.

2) HOW WOULD YOU IMPROVE THE RUN TIME WITHOUT CHANGING THE GAP SIZE?

WITHOUT THE GAPS, SHELL SORT CAN IMPROVE THROUGH ITS SWAPS AND INSERTIONS. USING A XOR SWAP MAY IMPROVE THE TIME MARGINALLY BECAUSE ALL OUR ELEMENTS ARE INTS

# QUICK SORT

-DIVIDE AND CONQUER

- PARTITIONS INTO TWO SUB ARRAYS BY SELECTING A PIVOT
  - LESS = LEFT GREATER = RIGHT
  - USES SUBROUTINE FOR THIS
    - RETURNS INDEX THAT INDICATES DIVISION BETWEEN PARTITIONED PORTIONS

- START PARTITION AROUND LAST ELEMENT

**PSUEDOCODE:** (FROM LAB POC)

```
q_sort( arr, left, right):

    if  left < right:
      - index = partition(arr, left, right)
        q_sort( arr, index, left-1)
        q_sort( arr, index+1, right)


partition( arr, left, right):

    - pivot = arr[left]
    - lo = left+1
    - hi = right

    while true:

        while lo <= hi and arr[hi] >= pivot:        # comparison
            - hi -= 1

        while lo <= hi and arr[lo] <= pivot:        # comparison
            - lo += 1

        if lo <= hi:
            - swap [arr[lo], arr[hi]]               # 3 moves
        else
            break

        swap ( arr[left], arr[hi])                  # 3 moves

        return hi
```

# PRE LAB 3

1) WHY ISN'T QUICKSORT DOOMED BY WORST CASE SCENARIO OF $O(n^2)$ ?

QUICKSORT IS NOT DOOMED BY ITS WORST CASE SCENARIO BECAUSE IT OCCURS IF ALL THE ELEMENTS ARE THE SAME, ALREADY SORTED, OR IN REVERSE IF YOU USE THE FIRST OR LAST ELEMENT AS THE PIVOT. USING A RANDOM VALUE IN THE ARRAY, OR THE MEDIAN WHICH CAN BE COMPUTED IN LINEAR TIME EITHER ELIMINATES OR MAKES THE $O(n^2)$ CASE INCREDIBLY RARE.

(USED INFO FROM pminaylov.com/sorting-algorithms/)

# BINARY INSERTION SORT.

- BINARY SEARCH TO FIND CORRECT LOCATION
  - INSERTS @ THAT POSITION.
- BINARY SEARCH SEARCHES BY MIDPOINTS → HIGHER LOWER
- FOR EACH ELE, RUN BINARY SEARCH THROUGH ELEMENTS TO THE LEFT

## PSUEDO CODE (LAB DOC)

```
bi_sort(arr):
  for (i=0, i<len(arr), i++)
    - val = arr[i], left=0, right= i
    - while left < right:
        - mid = left + ((right - left) / 2)   (FLOOR)
        - if val >= arr[mid]   (comparison)
            - left= mid +1
        - else
            - right = mid
    - for (j = i, j > left, i--)
        - swap(arr[j], arr[j-1])   3 moves
```

## PRE LAB 4

1) CAN YOU FIGURE OUT WHAT EFFECT THE BINARY SEARCH ALGORITHM HAS ON COMPLEXITY WHEN COMBINED WITH INSERTION.

THE PERFORMING OF BINARY INSERTION DOES $\log_2 n$ COMPARISONS AND THEREFORE IS $O(n \log n)$. HOWEVER THE PROCESS OF INSERTING ELEMENTS AND SHIFTING OTHER ELEMENTS KEEPS THE AVERAGE TIME COMPLEXITY $O(n^2)$.

(en.wikipedia.org/wiki/Insertion_sort)

# HELPER.H + .C

- FUNCTION FOR INITIALIZING ARRAY
  - calloc of uint32_t's
    - check if calloc worked
  - CALLS SRAND TO SET SEED
  - SETS EVERY VALUE IN ARRAY TO A RAND() VALUE MASKED BY 0x3FFFFFFF 0011 ... 11 32 bit
- FUNCTION TO PRINT ARRAY
- SWAP FUNCTION THAT ALSO INCREMENTS MOVE VARIABLE BY 3

# PRE LAB 5

1) HOW WILL YOU KEEP TRACK OF MOVES AND COMPARISONS

FOR MOST SORTS, I HAVE A LOCAL VARIABLE FOR COMPARISONS WHICH IS INCREMENTED IN THE SORT FUNCTION. I GET THIS VALUE WITH A GETTER FUNCTION, CALLED IN SORTING.C. FOR THE NUMBER OF MOVES, I RETURN IT IN THE SORTING FUNCTION ITSELF AND SET IT TO A VARIABLE IN SORTING.C. BECAUSE QUICKSORT IS RECURSIVE, I USED A GETTER FUNCTION FOR MOVES AS WELL.

## RANDOM NOTES:

- UTILITY FUNCTION FOR PRINTING ARRAY
- UTILITY FUNCTIONS FOR COMPARE + SWAP
  - FOR COMPARE USE STATIC VARIABLE IN THAT FILE AND HAVE A GET + SET METHOD TO CHANGE VARIABLES
  - could be modular = ITS OWN .H .C FILE
- SWAP = 3 MOVES
- COMPARISON OCCURS BETWEEN 2 ARRAY ELEMENTS
- BIT MASK RAND VALUES WHEN GENERATING / INITIALIZATION
- MASK BY 0011 ... 1 (32)