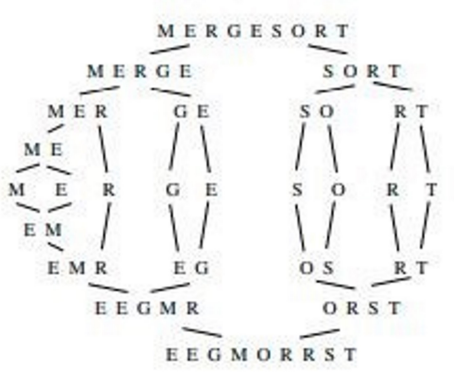
Sorting

15 July 2021 22:22

- **MERGE SORT:**
  - Recursive algorithm
  - Divide and Conquer
  - Using Queues to merge



Code:

```
mergesort(item_type s[], int low, int high) {
      int i; /* counter */
      int middle; /* index of middle element */
      if (low < high) {
            middle = (low+high)/2;
            mergesort(s,low,middle);
            mergesort(s,middle+1,high);
            merge(s, low, middle, high);
      }
}

merge(item_type s[], int low, int middle, int high){
      int i;     /* counter */
      queue buffer1, buffer2;
      /* buffers to hold elements for merging */

      init_queue(&buffer1);
      init_queue(&buffer2);
```

```
    for (i=low; i<=middle; i++) enqueue(&buffer1,s[i]);
    for (i=middle+1; i<=high; i++) enqueue(&buffer2,s[i]);

    i = low;
    while (!(empty_queue(&buffer1) ||
    empty_queue(&buffer2))) {
        if (headq(&buffer1) <= headq(&buffer2))
        s[i++] = dequeue(&buffer1);
        else
        s[i++] = dequeue(&buffer2);
    }

    /* If the queue sizes are different then there are
    elements that don't require comparing and rest
    elements are required to fill in the merge */
    while (!empty_queue(&buffer1))
        s[i++] = dequeue(&buffer1);
    while (!empty_queue(&buffer2))
        s[i++] = dequeue(&buffer2);
}
```

22:22

- **HEAP SORT:**
  - Priority queue requirement
  - Tree structure in heap
  - Methods include heapify, bubble_up, bubble_down, extract_minimum

Code:

```
pq_insert(priority_queue *q, item_type x){
    if (q->n >= PQ_SIZE)
        printf("Warning: priority queue overflow insert
        x=%d\n",x);
```

```
        else {
                q->n = (q->n) + 1;
                q->q[ q->n ] = x;
                bubble_up(q, q->n);
        }
}

/* This subroutine trickles the inserted node at the end of
the heap up to it's right position in the heap */
bubble_up(priority_queue *q, int p){
        if (pq_parent(p) == -1) return; /* at root of heap, no
        parent */
        if (q->q[pq_parent(p)] > q->q[p]) {
                pq_swap(q,p,pq_parent(p));
                bubble_up(q, pq_parent(p));
        }
}

make_heap(priority_queue *q, item_type s[], int n){
        int i; /* counter */
        pq_init(q);
        for (i=0; i<n; i++)
                pq_insert(q, s[i]);
}

item_type extract_min(priority_queue *q){
        int min = -1; /* minimum value */
        if (q->n <= 0)
                printf("Warning: empty priority queue.\n");
        else {
                min = q->q[1];
                q->q[1] = q->q[ q->n ];
                q->n = q->n - 1;
                bubble_down(q,1);
        }
        return(min);
}
```

```
bubble_down(priority_queue *q, int p){
    int c; /* child index */
    int i; /* counter */
    int min_index; /* index of lightest child */

    c = pq_young_child(p);
    min_index = p;

    /* This checks each child of a node in heap if they are
    more dominant than them and record the index */
    for (i=0; i<=1; i++)
        if ((c+i) <= q->n) {
            if (q->q[min_index] > q->q[c+i])
            min_index = c+i;
        }

    if (min_index != p) {
        pq_swap(q,p,min_index);
        bubble_down(q, min_index);
    }
}

heapsort(item_type s[], int n){
    int i; /* counters */
    priority_queue q; /* heap for heapsort */
    make_heap(&q,s,n);
    for (i=0; i<n; i++)
        s[i] = extract_min(&q);
}
```

- **QUICK SORT:**
  - Recursive algorithm
  - Divide and Conquer
  - Partitioning using a pivot index
  - Randomization algorithm

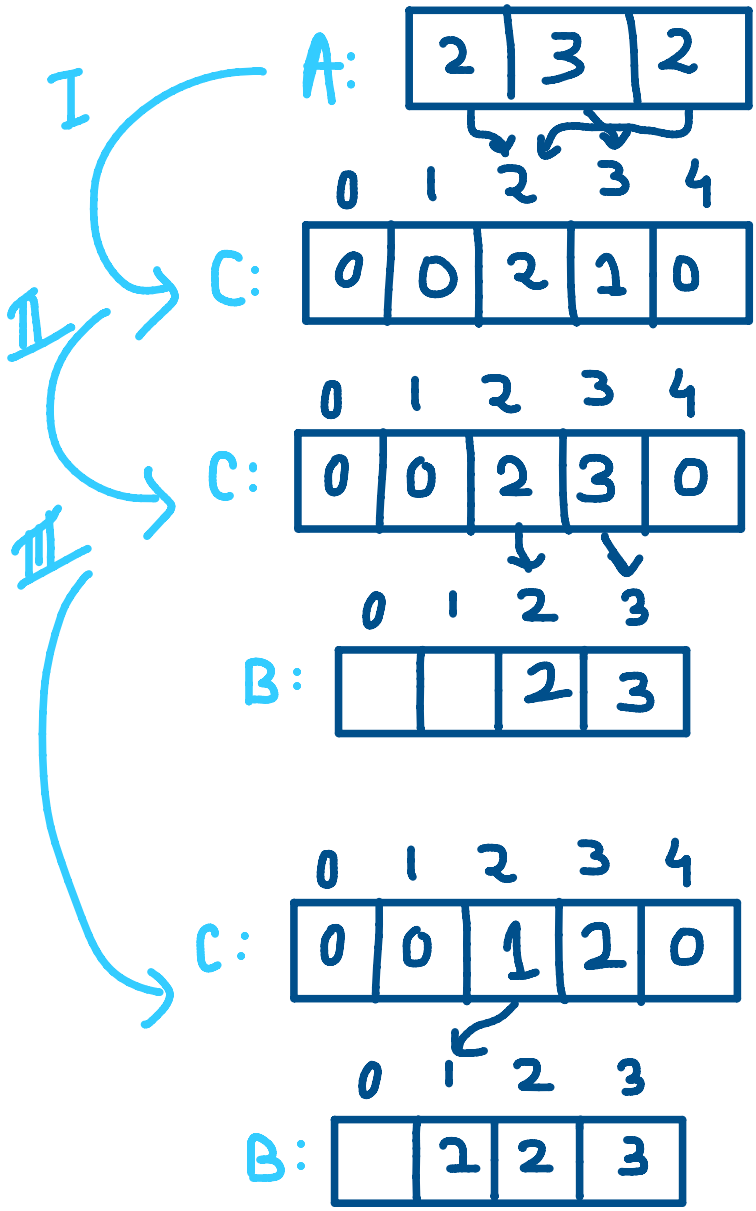
- Pivot element would be in the correct position, elements less than pivot would be before it and greater after it.

  Code:

```
quicksort(item_type s[], int l, int h){
      int p; /* index of partition */

      if ((h-l)>0) {
            p = partition(s,l,h);
            quicksort(s,l,p-1);
            quicksort(s,p+1,h);
      }
}

int partition(item_type s[], int l, int h){
      int i; /* counter */
      int p; /* pivot element index */
      int firsthigh; /* divider position for pivot element */

      p = h;
      firsthigh = l;

      for (i=l; i<h; i++)
            if (s[i] < s[p]) {
                  swap(&s[i],&s[firsthigh]);
                  firsthigh ++;
            }
      swap(&s[p],&s[firsthigh]);
      return(firsthigh);
}
```

- **COUNTING SORT:**
  - Sort assumes elements are in set of 0 to K
  - Requires an array index 0 to K as a mediatory array as well as an extra final array to store the final answer.

- No requirement for comparison for sorting



Code:

```
counting_sort(item_type A[], int k){
        item_type C[k+2];
        int i;
        for(i=0;i<=k;i++){
                C[i]=0;
        }
        for(i=0;i<k;i++){
                C[A[i]] = C[A[i]] + 1;
```

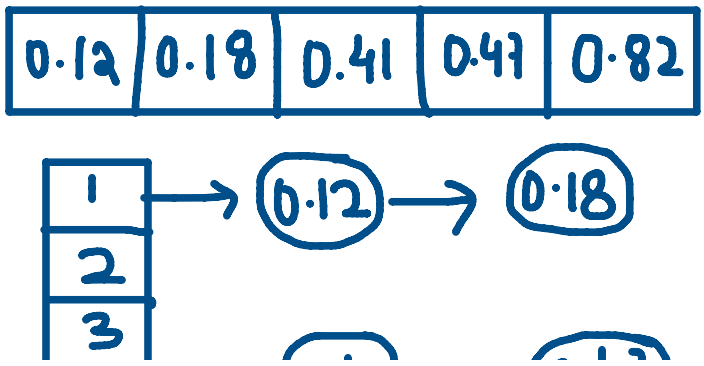
```
        }
        for(i=0;i<k;i++){
            C[i] = C[i] + C[i-1];
        }
        for(i=k;i>0;i--){
            B[C[A[i]] = C[A[i]];
            C[A[i]] = C[A[i]] - 1;
        }
    }
```

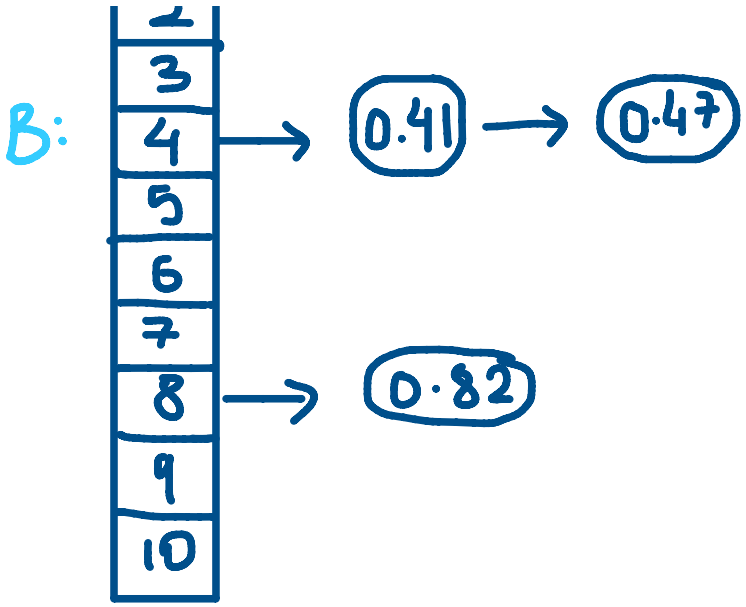
- **RADIX SORT:**
  - Sorts each digits sequentially

    Code:

```
radixsort(item_type A[], int l, int h){
    For all 1 to digits:
        Sort A on each digits
}
```

- **BUCKET SORT:**
  - Sorting algorithm appropriate for continuous values in range [0,1)
  - Uses array of linked lists as buckets in order to group elements in the same range and sort each bucket individually then combine





Code:

```
bucketsort(item_type A[], int n){
    Declare B[n] as new array
    For 0 to n-1:
        B[i] = empty list
    For 0 to n-1:
        insert A[i] to list B[floor(A[i])]
    For 0 to n-1:
        Sort list B[i]
    Concatenate all lists in B[0 to n-1]
}
```