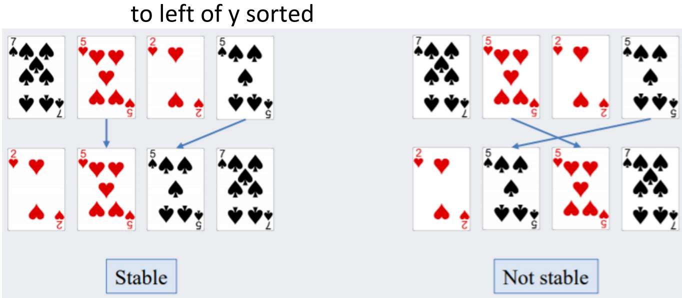
Sorting

November 18, 2019 8:13 AM

Sorting algorithms

- Computational Complexity
    - Avg case
    - worst/best case
- Memory usage
- Stability
    - maintains relative order of records with equal keys
    - for records x and y with equal keys, if x appears to left of y unsorted, x still appears to left of y sorted



Selection Sort

- repeatedly finds smallest item
- repeatedly swap first unsorted item with smallest unsorted item

int findMin (int arr[], int size) {
int minval, minIndex =0;
minIndex = 0;
minval = arr[];
for(int=1; i<size; i++) {
if (minimal>arr[i]) {
minVal=arr[i];
minIndex=i
}
}
return minIndex; //return minval
}

23	41	33	81	7	19	11	45	Find smallest unsorted item: 7 comparisons
7	41	33	81	23	19	11	45	Find smallest unsorted item: 6 comparisons
7	11	33	81	23	19	41	45	Find smallest unsorted item: 5 comparisons
7	11	19	81	23	33	41	45	Find smallest unsorted item: 4 comparisons
7	11	19	23	81	33	41	45	Find smallest unsorted item: 3 comparisons
7	11	19	23	33	81	41	45	Find smallest unsorted item: 2 comparisons
7	11	19	23	33	41	81	45	Find smallest unsorted item: 1 comparison
7	11	19	23	33	41	45	81	Sorted

- number of comparison operations

Unsorted elements	Comparisons
n	$n-1$
$n-1$	$n-2$
…	…
3	2
2	1
1	0
	$n(n-1)/2$

```
void selectionSort(int arr[], int size)
{
  int i; // next index to be set to minimum
  int min_pos; // index of minimum element
  for (i = 0; i < size-1; i++) {
    min_pos = minPosition(arr, i, size-1)
    if (min_pos != i)
      swap(&arr[min_pos], &arr[i]);
  }
}
```

```
int minPosition(int arr[], int start, int end)
{
  int min_pos = start;
  int j;
  for (j = start + 1; j <= end; j++) {
    if (arr[j] < arr[min_pos])
      min_pos = j;
  }
  return min_pos;
}
```

- Selection sort not stable
- Makes n*(n-1)/2 comparisons regardless of original order of input
- performs n-1 swaps: #write = O(n)
- run time O(n^2) - best/worst/avg

Insertion sort

- divides array into sorted and unsorted parts
- sorted part of array expanded one element at a time
- find correct place in sorted part to place 1st element of unsorted part
- find correct place in sorted part to place 1st element of unsorted
- move element after insertion point up one position to make place

```
void insertionSort(int arr[], int size)
{
  int i, temp, position;
```

First element is already "sorted"

23	41	33	81	7	19	11	45	Locate position for 41 – 1 comparison
23	41	33	81	7	19	11	45	Locate position for 33 – 2 comparisons
23	33	41	81	7	19	11	45	Locate position for 81 – 1 comparison
23	33	41	81	7	19	11	45	Locate position for 7 – 4 comparisons

- move element after insertion point up one position to make place

```c
void insertionSort(int arr[], int size)
{
  int i, temp, position;
  for (i = 1; i < size; i++)
  {
    temp = arr[i];
    position = i;
    // Shuffle up all sorted items > arr[i]
    while (position > 0 && arr[position - 1] > temp)
    {
      arr[position] = arr[position - 1];
      position--;
    }
    // Insert the current item
    arr[position] = temp;
  }
}
```

23	33	41	81	7	19	11	45	Locate position for 81 – 1 comparison
23	33	41	81	7	19	11	45	Locate position for 7 – 4 comparisons
7	23	33	41	81	19	11	45	Locate position for 19 – 5 comparisons
7	19	23	33	41	81	11	45	Locate position for 11 – 6 comparisons
7	11	19	23	33	41	81	45	Locate position for 45 – 2 comparisons
7	11	19	23	33	41	45	81	Sorted

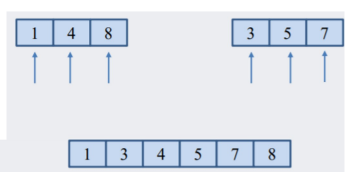
Sorted Elements	Worst-case Search	Worst-case Shuffle
0	0	0
1	1	1
2	2	2
…	…	…
$n-1$	$n-1$	$n-1$
	$n(n-1)/2$	$n(n-1)/2$

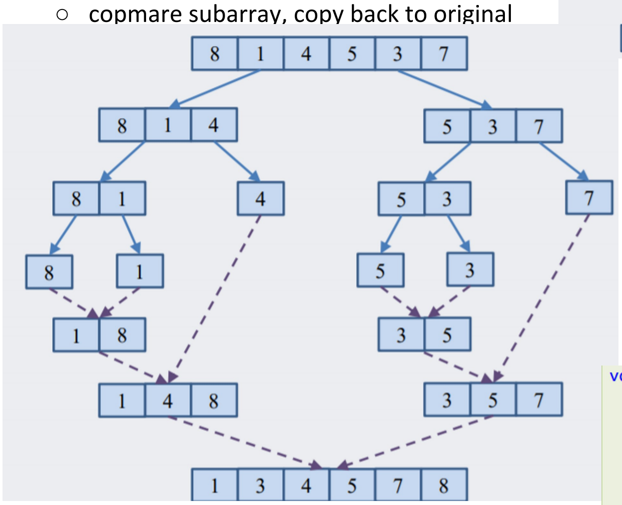
- Best case
  - affected by state of array to be sorted
  - in best case, array already sorted, n comparison
- Worst case
  - array in reverse order, every item moved to front
  - outer loop runs n-1 times, on avg n/2 comparisons
  - n*(n-1)/2 comparisons and moves
- Avg case
  - if random data sorted, insertion sort closer to worst case n*(n-1)/4

Name	Best	Average	Worst	Stable	Memory
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$

Merge sort

- splits problem into smaller subproblems, sloves, combines subproble solutions to form overall solution
- Split array into halves, and recursively sort each half
- Merge the two halves together to produce bigger sorted array, (sorting happens)
- O(m+n)
- merge steop copies subarray halves into temp array, divide arrays in half until each subarrray contains a single element
  - copmare subarray, copy back to original






```c
void msort(int arr[], int low, int high) {
  int mid;
  if (low < high) {
    // subarray has more than 1 element
    mid = (low + high) / 2;
    msort(arr, low, mid);
    msort(arr, mid+1, high);
    merge(arr, low, mid, high);
  }
}
```

```c
void mergeSort(int arr[], int size) {
  msort(arr, 0, size-1);
}
```

```c
void merge(int arr[], int low, int mid, int high) {
  int i = low, j = mid+1, index = 0;
  int* temp = (int*) malloc((high - low + 1) * sizeof(int));
  while (i <= mid && j <= high) {
    if (arr[i] <= arr[j])
      temp[index++] = arr[i++];
    else
      temp[index++] = arr[j++];
  }
  if (i > mid) {
    while (j <= high)
      temp[index++] = arr[j++];
  }
  else {
    while (i <= mid)
      temp[index++] = arr[i++];
  }
```


```c
void msort(int arr[], int
  int mid;
  if (low < high) {
    // subarray has more
    mid = (low + high) /
    msort(arr, low, mid);
    msort(arr, mid+1, hig
    merge(arr, low, mid,
  }
}
void mergeSort(int a
  msort(arr, 0, size
}
```


Merge sort stability

- Worst case: n-1, check every subarry inces
- Best case: n/2, reach end of subarray, copy rest
- subarray dividied $\log_2 n$ dividions to reach 1-element subarray
- external sorting is a term for a class of sorting that can handle massive datas sets taht do not fit in RAM

subarray

- external sorting is a term for a class of sorting that can handle massive datas sets taht do not fit in RAM

```
else {
  while (i <= mid)
    temp[index++] = arr[i++];
}
for (index = 0; index < high-low; index++)
  arr[low + index] = temp[index];
free(temp);
}
```

```
}
```

```
void mergeSort(int a
  msort(arr, 0, size
}
```

Name	Best	Average	Worst	Stable	Memory
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	$O(n)$

Analysing recursive functions

- base case T(1)
- reunning time of subproblems can be similarly expressed in terms of running times of subproblems
- determine number of substitution levels req to reach base

```
double arrMax(double arr[], int size, int start) {
  if (start == size - 1)
    return arr[start];
  else
    return max( arr[start], arrMax(arr, size, start + 1) );
}
```

$T(1) \leq b$

$T(n) \leq c + T(n-1)$

- Analysis

$T(n) \leq c + c + T(n-2)$

$T(n) \leq c + c + c + T(n-3)$

$T(n) \leq k \cdot c + T(n-k)$

$T(n) \leq (n-1) \cdot c + T(1) = (n-1) \cdot c + b$

- $T(n) \in O(n)$