CSC 344 – Algorithms and Complexity

Lecture #3 – Internal Sorting

What is the Brute Force Approach?

- A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved
- Examples:
  1. Computing a^n ($a > 0$, n a nonnegative integer)
  2. Computing $n!$
  3. Multiplying two matrices
  4. Searching for a key of a given value in a list

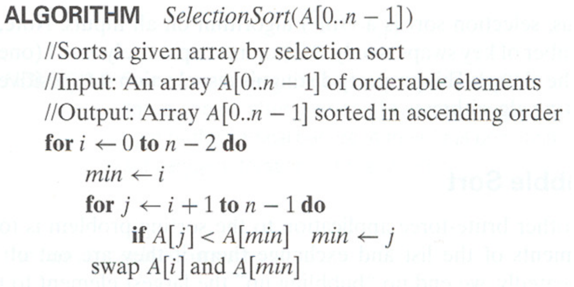
Brute-Force Sorting Algorithm

- Selection Sort
  - Scan the array to find its smallest element and swap it with the first element.
  - Starting with the second element, scan the elements after it to find the smallest among them and swap it with the second elements.
  - Generally, on pass i (0 ≤ i ≤ n-2), find the smallest element after A[i] and swap it with A[i]:
    $A[0] \leq \ . \ . \ . \leq A[i\text{-}1] \mid A[i], \ . \ . \ . \ . \ , A[min], . \ . \ ., A[n\text{-}1]$
    in their final positions
- Example: 7 3 2 5

Selection Sort

```
// selectionSort() - The selection sort where we
//                    seek the ith smallest value and
//                    swap it into its proper place
void  selectionSort(int x[], int n) {
      int   i, j, min;

      for (i = 0;  i < n-1; i++)    {
            min = i;
            for (j = i;  j < n; j++)      {
                  if (x[j] < x[min])
                        min = j;
                  swap(x[i], x[min]);
            }
      }
}
```

Analysis of Selection Sort



```
ALGORITHM SelectionSort(A[0..n − 1])
    //Sorts a given array by selection sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in ascending order
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[min]   min ← j
        swap A[i] and A[min]
```

- Time efficiency:
- Space efficiency:
- Stability:

Decrease and Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

3 Types of Decrease and Conquer

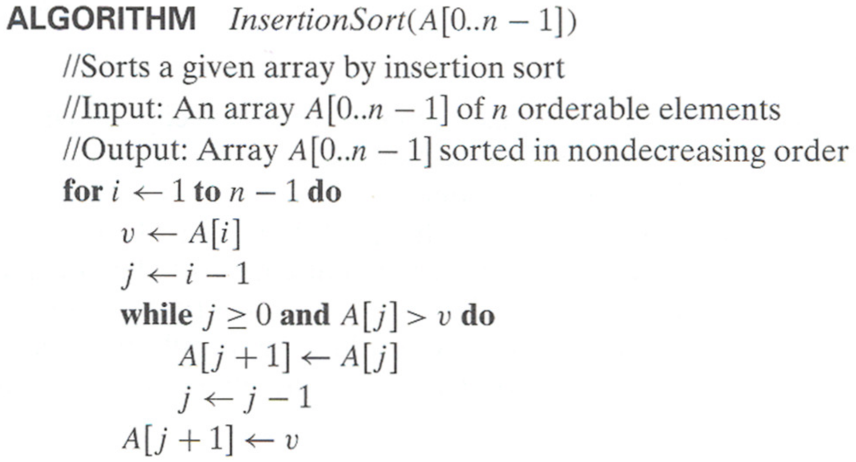
- Decrease by a constant (usually by 1):
  - insertion sort
- Decrease by a constant factor (usually by half)
  - binary search
  - exponentiation by squaring
- Variable-size decrease
  - Euclid's algorithm
  - Nim-like games

Insertion Sort

- To sort array A[0..n-1], sort A[0..n-2] recursively and then insert A[n-1] in its proper place among the sorted A[0..n-2]
- Usually implemented bottom up (nonrecursively)
- Example: Sort 6, 4, 1, 8, 5

6 | 4 1 8 5
4 6 | 1 8 5
1 4 6 | 8 5
1 4 6 8 | 5
1 4 5 6 8

Pseudocode of Insertion Sort



```
ALGORITHM InsertionSort($A[0..n-1]$)
    //Sorts a given array by insertion sort
    //Input: An array $A[0..n-1]$ of $n$ orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    for $i \leftarrow 1$ to $n-1$ do
        $v \leftarrow A[i]$
        $j \leftarrow i-1$
        while $j \geq 0$ and $A[j] > v$ do
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j-1$
        $A[j+1] \leftarrow v$
```

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n-1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost sorted arrays)

- Space efficiency: in-place
- Stability: yes
- Best elementary sorting algorithm overall

Insertion Sort

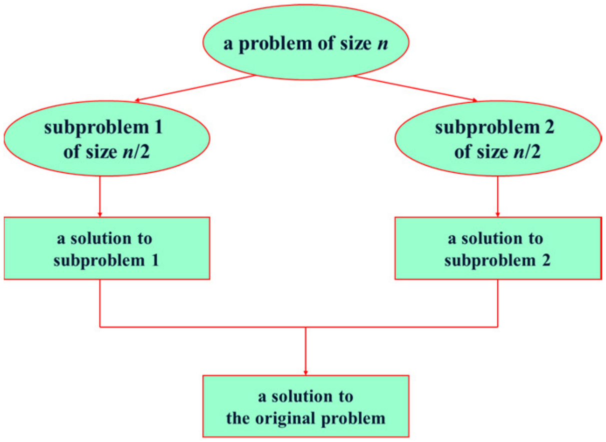
```
// insertionSort() - The insertion sort where we
//                   seek to insert the next value
//                   into the sorted portion of the
//                   array
void insertionSort(int x[], int n)  {
      int i, j;
      int   temp;

      // Insert the ith element into its
      // proper place
```

```
      for (i = 1;  i < n;  i++)     {
            // temp is the value to be inserted
            temp = x[i];
            j = i - 1;

            // Work our way from the end of the
            // sorted portion of the array to temp's
            // proper place
            while (j >= 0 && x[j] > temp) {
                  x[j+1] = x[j];
                  j = j - 1;
            }

            x[j+1] = temp;
      }
}
```

Divide and Conquer

- The most-well known algorithm design strategy:
  1. Divide instance of problem into two or more smaller instances
  2. Solve smaller instances recursively
  3. Obtain solution to original (larger) instance by combining these solutions

Divide and Conquer Technique (continued)



a problem of size *n*

subproblem 1 of size *n*/2

subproblem 2 of size *n*/2

a solution to subproblem 1

a solution to subproblem 2

a solution to the original problem

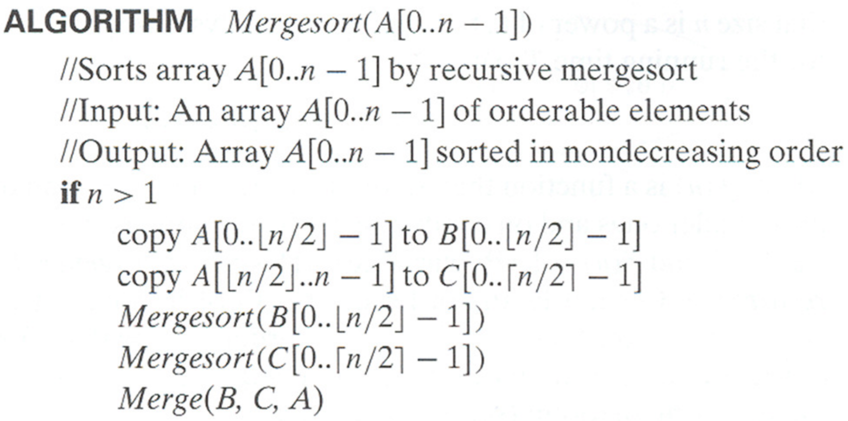
Divide and Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search: decrease-by-half

Mergesort

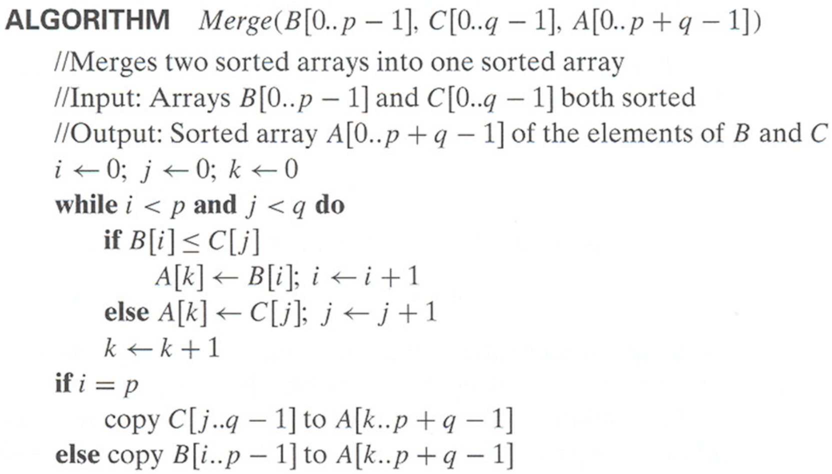
- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
  - Compare the first elements in the remaining unprocessed portions of the arrays
  - Copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort



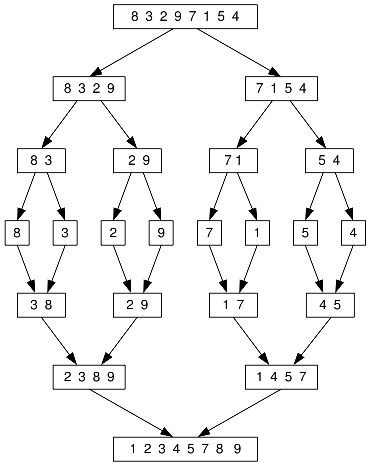
```
**ALGORITHM** Mergesort($A[0..n-1]$)
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)
        Mergesort($C[0..\lceil n/2 \rceil - 1]$)
        Merge($B, C, A$)
```

Pseudocode of Merge



```
ALGORITHM Merge($B[0..p-1]$, $C[0..q-1]$, $A[0..p+q-1]$)
    //Merges two sorted arrays into one sorted array
    //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
    //Output: Sorted array $A[0..p+q-1]$ of the elements of $B$ and $C$
    $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \le C[j]$
            $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else copy $B[i..p-1]$ to $A[k..p+q-1]$
```

Mergesort Example



Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

  $$\lceil \mathrm{log2}\ n! \rceil \approx n\ \mathrm{log2}\ n - 1.44n$$

- Space requirement: $\Theta(n)$ (not in-place)
- Can be implemented without recursion (bottom-up)

Mergesort

```
// mergeSort() - A recursive version of the merge
//               sort, where the array is divided
//               into smaller and smaller subarrays
//               and then merged together in order
void mergeSort(int x[], int n) {
      int   *y, *z;

      // If the subarrays are n't trivial (size = 1)
      // divide them into two subarrays, sort them
      // using the merge sort and then merge them
      // back together
```

```
      if (n > 1)  {
            //  Set up arrays of the required size
            y = new int[n/2];
            z = new int[n/2];

            // Copy the first half of x into y
            for (int i = 0;  i < n/2; i++)
                  y[i] = x[i];

            // Copy the second half of x into z
            for (int i = n/2, j = 0; i < n; i++)
                  z[j++] = x[i];
```

```
            // Sort y and z and then merge them
            mergeSort(y, n/2);
            mergeSort(z, n/2);
            merge(y, n/2, z, n/2, x, n);
      }
}
```

Merge

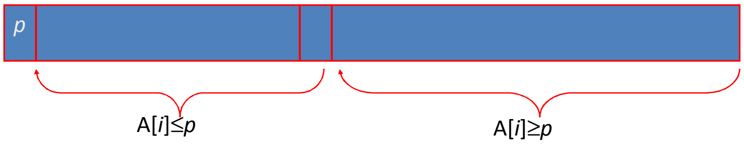
```
// merge() - Merge the two subarrays together
//                 Maintaining the order
void merge(int b[], int bSize,
                  int c[], int cSize,
                  int a[], int aSize) {
      int i = 0, j = 0, k = 0;

      // As long as there are still unmerged
      // values in both subarrays
      while (i < bSize && j < cSize)      {
            // The smaller of the two values is
            // copied back into the larger array
```

```
        if (b[i] <= c[j])
              a[k] = b[i++];
        else
              a[k] = c[j++];
        k = k + 1;

        // Copy back the remainder of the
        // subarray that still has values that
        // were not yet copied back
```

```
        if (i == bSize)   {
              for (int m = j;  m < cSize; m++)
                    a[bSize+m] = c[m];
        }
        else  {
              for (int m = i;  m < bSize; m++)
                    a[cSize+m] = b[m];
        }
  }
```

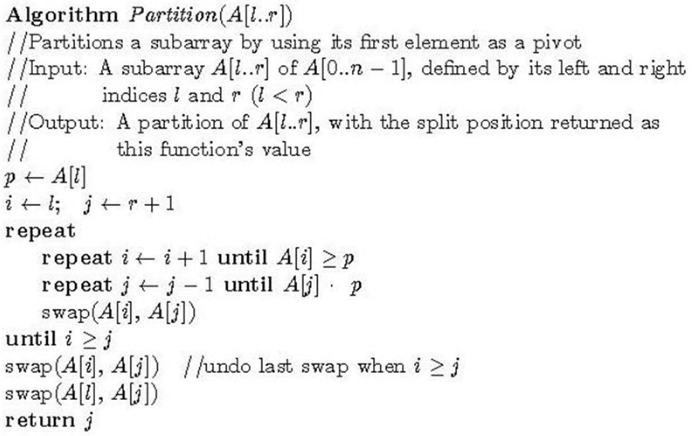
Quicksort

- Select a ***pivot*** (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Hoare's Partitioning Algorithm

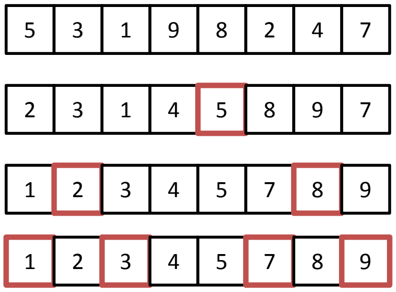


```
Algorithm Partition($A[l..r]$)
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
//        indices $l$ and $r$ ($l < r$)
//Output: A partition of $A[l..r]$, with the split position returned as
//        this function's value
$p \leftarrow A[l]$
$i \leftarrow l$;   $j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ until $A[i] \geq p$
    repeat $j \leftarrow j - 1$ until $A[j] \cdot p$
    swap($A[i]$, $A[j]$)
until $i \geq j$
swap($A[i]$, $A[j]$)   //undo last swap when $i \geq j$
swap($A[l]$, $A[j]$)
return $j$
```

Quicksort Example

- 5 3 1 9 8 2 4 7

Quicksort Example



5	3	1	9	8	2	4	7
2	3	1	4	5	8	9	7
1	2	3	4	5	7	8	9
1	2	3	4	5	7	8	9

QuickSort

```
// quickSort() - Call the recursive quick
//                       sort method
void  quickSort(int x[], int n)     {
      quick(x, 0, n-1);
}
```

Quick

```
// quick() - Place the pivot in its proper
//           place and recursive sort
//           every on either side of it
void  quick(int x[], int low, int high)   {
      int pivotPlace;

      if (low >= high)
            return;

      pivotPlace = partition(x, low, high);
      quick(x, low, pivotPlace-1);
      quick(x, pivotPlace+1, high);
}
```

Partition

```
// partition() - reaarange the array so that the
//               first value in this portion of the
//               array is in its proper place and
//               every other element is on the
//               correct side of that value
int partition(int x[], int low, int high) {
      int   pivot;
      int   i, j;

      // The lowest value in this portion
      // of the array is the pivot
      // and we start rearranging from
      // this portion's lower and upper
      // bounds
```

```
      pivot = x[low];
      i = low;
      j = high+1;

      // We keep moving forward and backward until
      // we have another pair of elements to be
      // moved
      do    {
            do    {
                  i = i + 1;
            } while (x[i] <= pivot);

            do    {
                  j = j - 1;
            } while (x[j] > pivot);
```

```
            swap(x[i], x[j]);
      }  while (i < j);

      // We undo the last swap and swap
      // the pivot in its proper place
      swap(x[i], x[j]);
      swap(x[low], x[j]);

      return j;
}
```

Swap

```
// swap() - Swap the two parameter's values
void  swap(int &a, int &b) {
      int   temp;

      temp = a;
      a = b;
      b = temp;
}
```

Analysis of Quicksort

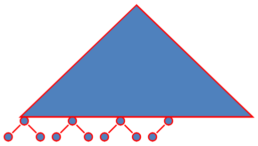
- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n2)$
- Average case: random arrays — $\Theta(n \log n)$

Analysis of Quicksort

- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion
- These combine to 20-25% improvement
- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

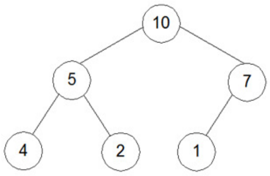
Heaps and Heapsort

- Definition - A ***heap*** is a binary tree with keys at its nodes (one key per node) such that:
- It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing

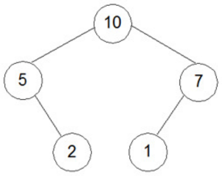


- The key at each node is ≥ keys at its children

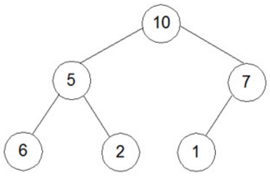
Which One is a Valid Heap?



a heap



not a heap



not a heap

- NB: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right

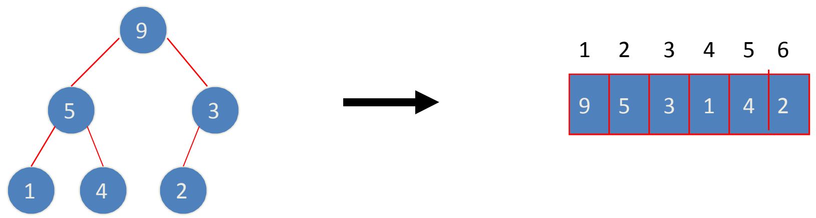
Some Important Properties of a Heap

- Given n, there exists a unique binary tree with n nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be represented as an array

Heap's Array Representation

Store heap's elements in an array (whose elements indexed, for convenience, 1 to n) in top-down left-to-right order

Example:



1	2	3	4	5	6
9	5	3	1	4	2

- Left child of node j is at $2j$
- Right child of node j is at $2j+1$
- Parent of node j is at $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations

Heap Construction (Bottom-Up)

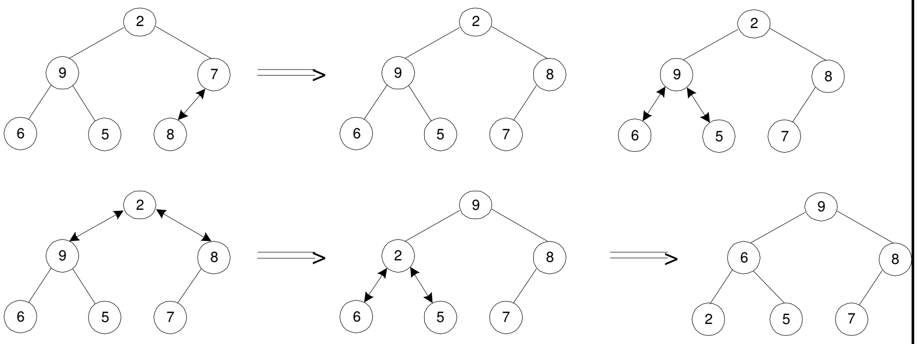
Step 0: Initialize the structure with keys in the order given

Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds

Step 2: Repeat Step 1 for the preceding parental node

Example of Heap Construction

- Construct a heap for the list 2, 9, 7, 6, 5, 8



Heapsort

- **Stage 1**: Construct a heap for a given list of n keys
- **Stage 2**: Repeat operation of root removal n-1 times:
  - Exchange keys in the root and in the last (rightmost) leaf
  - Decrease heap size by 1
  - If necessary, swap new root with larger child until the heap condition holds

Pseudocode of Sift Down

```
From (ni-2)/2 down to 0:
    1. if the parent is less than one child or bother
       children:
        a. swap the parent with the greater of the two children
```

Example of Sorting by Heapsort

- Sort the list 2, 9, 7, 6, 5, 8 by Heapsort

Stage 1 (heap construction)	Stage 2 (root/max removal)
2 9 $\underline{\mathbf{7}}$ 6 5 8	$\underline{\mathbf{9}}$ 6 8 2 5 7
2 $\underline{\mathbf{9}}$ 8 6 5 7	7 6 8 2 5 | 9
$\underline{\mathbf{2}}$ 9 8 6 5 7	$\underline{\mathbf{8}}$ 6 7 2 5 | 9
9 $\underline{\mathbf{2}}$ 8 6 5 7	5 6 7 2 | 8 9
9 6 8 2 5 7	$\underline{\mathbf{7}}$ 6 5 2 | 8 9
	2 6 5 | 7 8 9
	$\underline{\mathbf{6}}$ 2 5 | 7 8 9
	5 2 | 6 7 8 9
	$\underline{\mathbf{5}}$ 2 | 6 7 8 9
	2 | 5 6 7 8 9

Analysis of Heapsort

- Stage 1: Build heap for a given list of n keys worst-case

$$C(n) = \sum_{i=0} 2(h-i)\,\underset{\#\ nodes\ at\ level\ i}{\underset{\uparrow}{2^i}} \quad = \quad 2\,(\,n - \log_2(n+1)) \;\in\; \Theta(n)$$

- Stage 2: Repeat operation of root removal n-1 times (fix heap) worst-case

$$\mathrm{C(n)} = \sum_{i=1}^{n-1} 2\log_2 i \in \Theta(n\log n)$$

Both worst-case and average-case efficiency: $\Theta(n\log n)$

In-place: yes

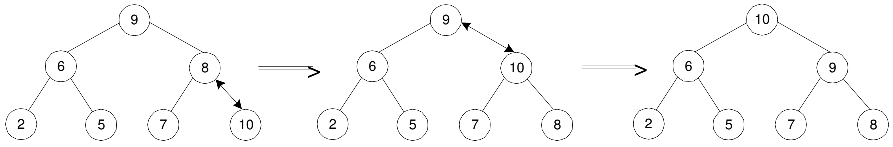
Stability: no (e.g., 1 1)

Priority Queue

- A *priority queue* is the ADT of a set of elements with numerical priorities with the following operations:
  - Find element with highest priority
  - Delete element with highest priority
  - Insert element with assigned priority (see below)
- Heap is a very efficient way for implementing priority queues
- Two ways to handle priority queue in which highest priority = smallest number

Insertion of a New Element into a Heap

- Insert the new element at last position in heap.
- Compare it with its parent and, if it violates heap condition, exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied
- Example: Insert key 10
- Efficiency: O(log *n*)



Heapsort

```
void heapSort(int *a, int count)
{
    int start, end;

    /*
     * heapify - Rearrange the element so that the
     *    father's value is greater than either son
     */
    for (start = (count-2)/2; start >=0; start--) {
        siftDown( a, start, count);
    }
```

```
    /* Swap the top element into its proper place
             and heapify again  */
    for (end=count-1; end > 0; end--) {
        swap(a[end],a[0]);
        siftDown(a, 0, end);
    }
}
```

Sift Down

```
void siftDown(int *a, int start, int end)
{
    int root = start;

    while ( root*2+1 < end ) {
        int child = 2*root + 1;
        if ((child + 1 < end)
                  && (a[child] < a[child+1])) {
            child += 1;
        }
```

```
        if (a[root] < a[child]) {
            swap( a[child], a[root] );
            root = child;
        }
        else
            return;
    }
}
```

Bubble Sort

- In a bubble sort, we compare adjacent element to determine if they are in order with respect to each other. If they aren't, we swap them.
- The process is repeated until we can pass through the entire array without needing to swap any elements.
- In theory, this should be more efficient than a selection sort because you don't need to pass through the array more than once if it is alrady in order.
- In practice, this is not necessarily the case; it requires a lot of data moves to place any data item in its proper position, and it is highly depend on the original order of the data items and the direction of the scan.

Bubble Sort

```
// bubbleSort() - A bubble sort function
void bubbleSort(int x[], int n) {
      bool switched;  // Have we switched them
                      // this time?

      int i = 0, j; // i counts the number of times
                    // through the array
                    // j keeps track of which
                    // element we're up to

      int temp; // Holds the value being
                // swapped
```

```
      do {
            // We haven't swapped anything yet
            switched = false;

            // Go through the array and see if any
            // two adjacent elements are out of
            // order
            for (j = 0; j < n - i - 1; j++)
                  if (x[j] > x[j+1]) {
                        // If so, swap them
                        switched = true;
                        temp = x[j];
                        x[j] = x[j+1];
                        x[j+1] = temp;
                  }
```

```
                  // Count passes through the array
                  // They shouldn't exceed n
                  i++;

                  // If we go a pass without a swap
                  // we're finished
                  // If we go through n passes we're
                  // finished
      } while (i < n && switched);
}
```

Data for a Bubble Sort Example

- 25 13 4 29 14 1 31 18

Tracing the Bubble Sort

Original	25	13	4	29	14	1	31	18
After Pass 1	13	4	25	14	1	29	18	31
After Pass 2	4	13	14	1	25	18	29	31
After Pass 3	4	13	1	14	18	25	29	31
After Pass 4	4	1	13	14	18	25	29	31
After Pass 5	1	4	13	14	18	25	29	31
After Pass 6	1	4	13	14	18	25	29	31

Final scan to confirm its in order

Bubble Sort vs Cocktail Shaker Sort

- A large value at the beginning of the array will move all the way to the end in one pass if the scan goes from beginning to end.
- A small value at the end of the array will move all the way to the beginning in one pass if the scan goes from end to beginning
- In both cases the reverse will require *n* passes.
- By scanning from beginning to end and then end to beginning removes this dependence. We call such a sort the ***cocktail shaker sort***.

Cocktail Shaker Sort

```
void  cocktailShakerSort(int x[], int n)  {
      bool switched;    // Have we switched them
                      // this time?

      int i = 0, j; // i counts the number of times
                    // through the array
                    // j keeps track of which
                    // element we're up to

      int temp; // Holds the value being
                // swapped
```

```
do {
      // We haven't swapped anything yet
      switched = false;

      // First we bubble down
      for (j = 0; j < n - i - 1; j++)
            if (x[j] > x[j+1]) {
                  // If so, swap them
                  switched = true;
                  temp = x[j];
                  x[j] = x[j+1];
                  x[j+1] = temp;
            }
```

```
      // Then we bubble up
      for (j = n - i - 2; j > 0; --j)
            if (x[j] > x[j+1]) {
                  // If so, swap them
                  switched = true;
                  temp = x[j];
                  x[j] = x[j+1];
                  x[j+1] = temp;
            }

            // Count passes through the array
            // They shouldn't exceed n
            i++;
```

```
                    // If we go a pass without a swap
                    // we're finished
                    // If we go through n passes we're
                    // finished
          } while (i < n && switched);

     }
```

Tracing the Cocktail Shaker Sort

Original	25	13	4	29	14	1	31	18
After Pass 1a	13	4	25	14	1	29	18	31
After Pass 1b	13	1	4	25	14	18	29	31
After Pass 2a	1	4	13	14	18	25	29	31
After Pass 2b	1	4	13	14	18	25	29	31

Final scan to confirm its in order