Searching, Sorting

part 1
Week 3 Objectives

- Searching: binary search
- Comparison-based search: running time bound
- Sorting: bubble, selection, insertion, merge
- Sorting: Heapsort
- Comparison-based sorting time bound
Brute force/linear search

- Linear search: look through all values of the array until the desired value/event/condition found
- Running Time: linear in the number of elements, call it $O(n)$
- Advantage: in most situations, array does not have to be sorted
Binary Search

- Array must be sorted
- Search array \( A \) from index \( b \) to index \( e \) for value \( V \)
- Look for value \( V \) in the middle index \( m = (b+e)/2 \)
  - That is compare \( V \) with \( A[m] \); if equal return index \( m \)
  - If \( V < A[m] \) search the first half of the array
  - If \( V > A[m] \) search the second half of the array

\[
\begin{array}{cccccccc}
V=3 & -4 & -1 & 0 & 0 & 1 & 1 & 3 & 19 & 29 & 47 \\
A[m]=1 < V=3 \Rightarrow & \text{search moves to the right half}
\end{array}
\]
Binary Search Efficiency

• every iteration/recursion
  - ends the procedure if value is found
  - if not, reduces the problem size (search space) by half

• worst case: value is not found until problem size=1
  - how many reductions have been done?
  - \( n / 2 / 2 / 2 / \ldots / 2 = 1 \). How many 2-s do I need?
  - if \( k \) 2-s, then \( n = 2^k \), so \( k \) is about \( \log(n) \)
  - worst running time is \( O(\log n) \)
Search: tree of comparisons

- Tree of comparisons: essentially what the algorithm does
Search: tree of comparisons

- tree of comparisons: essentially what the algorithm does
  - each program execution follows a certain path
Search: tree of comparisons

- tree of comparisons: essentially what the algorithm does
  - each program execution follows a certain path
  - red nodes are terminal / output
  - the algorithm has to have at least n output nodes... why?
Search: tree of comparisons

- tree of comparisons: essentially what the algorithm does
  - each program execution follows a certain path
  - red nodes are terminal / output
  - the algorithm has to have n output nodes... why?
  - if tree is balanced, longest path = tree depth = log(n)
**Bubble Sort**

- **Simple idea:** as long as there is an inversion, swap the bubble
  - inversion = a pair of indices $i < j$ with $A[i] > A[j]$
  - swap $A[i] \leftrightarrow A[j]$
    - directly swap $(A[i], A[j])$

- **how long does it take?**
  - worst case: how many inversions have to be swapped?
  - $O(n^2)$
Insertion Sort

- partial array is sorted

```
1  5  8  20  49
```

- get a new element \( V = 9 \)
Insertion Sort

- partial array is sorted

  1  5  8  20  49

- get a new element $V=9$

- find correct position with binary search $i=3$
Insertion Sort

- Partial array is sorted

| 1 | 5 | 8 | 20 | 49 |

- Get a new element $v=9$

- Find correct position with binary search $i=3$

- Move elements to make space for the new element

| 1 | 5 | 8 | 20 | 49 |
Insertion Sort

- partial array is sorted

\[
\begin{array}{cccccc}
1 & 5 & 8 & 20 & 49 & \\
\end{array}
\]

- get a new element \( V = 9 \)

- find correct position with binary search \( i = 3 \)

- move elements to make space for the new element

\[
\begin{array}{cccccc}
1 & 5 & 8 & 20 & 49 & \\
\end{array}
\]

- insert into the existing array at correct position

\[
\begin{array}{cccccc}
1 & 5 & 8 & 9 & 20 & 49 \\
\end{array}
\]
Insertion Sort - variant

- partial array is sorted

| 1 | 5 | 8 | 20 | 49 |
Insertion Sort – variant

- partial array is sorted

1 5 8 20 49
Insertion Sort – variant

- partial array is sorted

  \[
  \begin{array}{lllll}
  1 & 5 & 8 & 20 & 49 \\
  \end{array}
  \]

- get a new element \( V = 9 \); put it at the end of the array

  \[
  \begin{array}{llllll}
  1 & 5 & 8 & 20 & 49 & 9 \\
  \end{array}
  \]
Insertion Sort – variant

- Partial array is sorted

| 1 | 5 | 8 | 20 | 49 |

- Get a new element $V=9$; put it at the end of the array

| 1 | 5 | 8 | 20 | 49 | 9 |

- Move in $V=9$ from the back until reaches correct position

| 1 | 5 | 8 | 20 | 9 | 49 |
Insertion Sort – variant

- partial array is sorted

| 1 | 5 | 8 | 20 | 49 |  |  |

- get a new element \( V=9 \); put it at the end of the array

| 1 | 5 | 8 | 20 | 49 | 9 |  |  |

- Move in \( V=9 \) from the back until reaches correct position

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>8</th>
<th>20</th>
<th>9</th>
<th>49</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>20</td>
<td>49</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Insertion Sort Running Time

- For one element, there might be required to move $O(n)$ elements (worst case $\Theta(n)$)
  - $O(n)$ insertion time

- Repeat insertion for each element of the $n$ elements gives $n \times O(n) = O(n^2)$ running time
Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again

Insertion/Selection
Running Time = $O(n^2)$
## Selection Sort

- **sort** array \( A[] \) into a new array \( C[] \)

- **while** (condition)
  - find **minimum** element \( x \) in \( A \) at index \( i \), ignore "used" elements
  - write \( x \) in next available position in \( C \)
  - mark index \( i \) in \( A \) as "used" so it doesn't get picked up again

- **Running Time** = \( O(n^2) \)
Selection Sort

- sort array \( A[\] \) into a new array \( C[\] \)

- while (condition)
  - find minimum element \( x \) in \( A \) at index \( i \), ignore "used" elements
  - write \( x \) in next available position in \( C \)
  - mark index \( i \) in \( A \) as "used" so it doesn't get picked up again

- Running Time = \( O(n^2) \)
Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$
Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$
**Selection Sort**

- sort array `A[]` into a new array `C[]`
- **while** (condition)
  - find **minimum** element `x` in `A` at index `i`, ignore "used" elements
  - write `x` in next available position in `C`
  - mark index `i` in `A` as "used" so it doesn't get picked up again

- **Running Time =** $O(n^2)$

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-5</td>
<td>-1</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-5</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
</tr>
</tbody>
</table>

The table shows the process of selecting minimum elements and placing them in a new array `C[]` while marking them as "used" in `A[]`. The running time complexity is $O(n^2)$. 
Selection Sort

- sort array A into a new array C

- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again

- Running Time = $O(n^2)$
Merge two sorted arrays

- **two sorted arrays**
  - $A[] = \{1, 5, 10, 100, 200, 300\};$ $B[] = \{2, 5, 6, 10\};$

- **merge them into a new array $C$**
  - index $i$ for array $A[]$, $j$ for $B[]$, $k$ for $C[]$
  - init $i=j=k=0$;
  - while (what_condition?)
    - if ($A[i] \leq B[j]$) {$C[k]=A[i], i++$} //advance $i$ in $A$
    - else {$C[k]=B[j], j++$} // advance $j$ in $B$
    - advance $k$
  - end_while
Merge two sorted arrays

complete pseudocode

index i for array A[], j for B[], k for C[]
init i=j=k=0;
while (k < size(A)+size(B)+1)
  if(i>size(A)) {C[k]=B[j], j++} // copy elem from B
  else if (j>size(B)) {C[k]=A[i], i++} // copy elem from A
  else if (A[i] <= B[j]) { C[k]=A[i], i++ } //advance i
  else {C[k]=B[j], j++} // advance j
  k++ //advance k
end_while
MergeSort

- divide and conquer strategy

- **MergeSort** array A
  - divide array A into two halves A-left, A-right
  - MergeSort A-left (recursive call)
  - MergeSort A-right (recursive call)
  - Merge (A-left, A-right) into a fully sorted array

- running time: $O(n \log n)$
MergeSort running time

- $T(n) = 2T(n/2) + \Theta(n)$
  - 2 sub-problems of size $n/2$ each, and a linear time to combine results
  - Master Theorem case 2 ($a=2$, $b=2$, $c=1$)
  - Running time $T(n) = \Theta(n \log n)$
Heap Data Structure

- binary tree
- max-heap property: parent > children
Max Heap property

- Assume the Left and Right subtrees satisfy the Max-Heap property, but the top node does not.

- Float down the node by consecutively swapping it with higher nodes below it.
Building a heap

- Representing the heap as array datastructure
  - Parent(i) = i/2
  - Left_child(i) = 2i
  - Right_child(i) = 2i+1

- A = input array has the last half elements leaves

- MAX-HEAPIFY the first half of A, reverse order

  for i=size(A)/2 downto 1
  MAX-HEAPIFY (A, i)
Heapsort

• Build a Max-Heap from input array

• LOOP
  – swap heap_root (max) with a leaf
  – output (take out) the max element; reduce size
  – MAX-HEAPIFY from the root to maintain the heap property

• END LOOP

• the output is in order
HeapSort running time

- Max-Heapify procedure time is given by recurrence
  - $T(n) \leq T(2n/3) + \Theta(1)$
  - master Theorem $T(n) = O(\log n)$
- Build Max-Heap: running $n$ times the Max-Heapify procedure gives the running time $O(n \log n)$
- Extracting values: again run $n$ times the Max-Heapify procedure gives the running time $O(n \log n)$
- Total $O(n \log n)$
tree of comparisons: essentially what the algorithm does

- each program execution follows a certain path
- red nodes are terminal / output
- the algorithm has to have n! output nodes... why?
- if tree is balanced, longest path = tree depth = n log(n)
QuickSort - pseudocode

QuickSort(A, b, e)  // array A , sort between indices b and e
- q = Partition(A, b, e) // returns pivot q, b<=q<=e
  // Partition also rearranges A so that if i<q then A[i] <= A[q]
  // and if i>q then A[i] >= A[q]
- if(b<q-1) QuickSort(A, b, q-1)
- if(q+1<e) QuickSort(A, q+1, e)

After Partition the pivot index contains the right value:

<table>
<thead>
<tr>
<th>b=0</th>
<th>q=3</th>
<th>e=9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
QuickSort Partition

● TASK: rearrange A and find pivot q, such that
  - all elements before q are smaller than A[q]
  - all elements after q are bigger than A[q]

● Partition (A, b, e)
  - x=A[e] //pivot value
  - i=b-1
  - for j=b TO e-1
    - if A[j]<=x then
  - swap A[i+1]<->A[e]
  - q=i+1; return q
Partition Example

- set pivot value \( x = A[e] \), // \( x=4 \)
  - \( i = \) index of last value < \( x \)
  - \( i+1 = \) index of first value > \( x \)

- run \( j \) through array indices \( b \) to \( e-1 \)
  - if \( A[j] \leq x \) //see steps (d), (e)
    - swap \( (A[j], A[i+1]) \);
    - \( i++ \); //advance \( i \)

- move pivot in the right place
  - swap \( (\text{pivot}=A[e], A[i+1]) \)

- return pivot index
  - return \( i+1 \)
QuickSort time

- Partition runs in linear time
  - If pivot position is q, the QuickSort recurrence is $T(n) = n + T(q) + T(n-q)$

- Best case q is always in the middle
  - $T(n)=n+2T(n/2)$, overall $\Theta(n\log n)$

- Worst case: q is always at extreme, 1 or n
  - $T(n) = n + T(1) + T(n-1)$, overall $\Theta(n^2)$
QuickSort Running Time

- Depends on the Partition balance

- Worst case: Partition produces unbalanced split $n = (1, n-1)$ most of the time
  - results in $O(n^2)$ running time

- Average case: most of the time split balance is not worse than $n = (cn, (1-c)n)$ for a fixed $c$
  - for example $c=0.99$ means balance not worse than $(1/100*n, 99/100*n)$
  - results in $O(n\log n)$ running time
  - can prove that on expectation (average), if pivot value is chosen randomly, running time is $\Theta(n\log n)$, see book.
Task: find k-th element
- k=n is same as “find MAX”, or “find highest”
- k=2 means “find second-smallest”
- k=1 is same as “finding MIN”

naive approach, based on selection sort:
- find first smallest (MIN)
- then find second smallest, third smallest, etc; until the k-th smallest element
- Running Time: average case k=Θ(n), and each “finding” min takes Θ(n) time, so total Θ(n²)
Median Stats

- “find k-th element”
- better approach, based on QuickSort
- Median(A,b,e,k) //find k-th greatest in array A, sort between indices b=1 and e=n
  
  - q = Partition(A,b,e) //returns pivot index q, b<=q<=e
  
  - //Partition also rearranges A so that if i<q then A[i] <= A[q]
  
  - // and if i>q then A[i] >= A[q]

  - if(q==k) return A[q] //found the k-th greatest
  - else Median(A,q+1,e,q-k)

- Not like Quicksort, Median recursion goes only on one side, depending on the pivot

- why the second Median call has k_{new}=q-k_{old}?
Median Stats

- **Running Time of Median**
- the recursive calls makes $T(n) = n + \max(T(q), T(n-q))$
  - "max": assuming the recursion has to call the longer side
  - just like QuickSort, average case is when $q$ is "balanced", i.e. $cn < q < (1-c)n$ for some constant $0 < c < 1$
  - balanced case: $T(n) = n + T(cn)$; Master Theorem gives linear time $\Theta(n)$
  - expected (average) case can be proven linear time (see book); worst case $\Theta(n^2)$
- worst case can run in linear time with a rather complicated choice of the pivot value before each partition call (see book)
Linear-time Sorting: Counting Sort

• Counting Sort (A[]) : count values, NO comparisons

• STEP 1 : build array C that counts A values
  - init C[]=0 ;
  - run index i through A
  - value = A[i]
  - C[value] ++; //counts each value occurrence

• STEP 2: assign values to counted positions
  - init position=0;
  - for value=0:RANGE
    - for i=1:C[value]
      - position = position+1;
  - OUTPUT[position]=value;
Counting Sort

- n elements with values in k-range of \{v_1,v_2,\ldots,v_k\}
  - for example: 100,000 people sorted by age: n=100,000; k = \{1,2,3,\ldots,170\} since 170 is maximum reasonable age in years.

- Linear Time \Theta(n+k)
  - Beats the bound? YES, linear \Theta(n), not \Theta(n*\log n), if k is a constant
  - Definitely appropriate when k is constant or increases very slowly
  - Not good when k can be large. Example: sort pictures by their size; n=10000 (typical picture collection), size range k can be any number from 200Bytes to 40MBytes.

- Stable (equal input elements preserve original order)