

Module 2

**Analysis of Algorithms

- Modularity
- Correctness
- Maintainability
- Functionality
- Robustness
- User-friendliness
- Programmer time
- Simplicity
- Extensibility
- Reliability

Sorting Algorithms

- Comparison
- Iterative
- Selection sort
- Bubble sort
- Insertion sort
- Recursive
- Merge sort
- Quick sort
- Heap sort
- Non-comparison
- Radix sort

Applications of Sorting Algorithms

- Uniqueness testing
- Deleting duplicates
- Prioritizing events
- Frequency counting
- Reconstructing the original order
- Set intersection/union
- Finding a target pair x, y such that $x+y=z$
- Efficient searching

Selection Sort

- At each step k , put the k th largest item in its place
- Given an array of n items
- Find the largest item l in the range of $[0 \dots n-1]$
- Swap l with the $(n-1)$ th item
- Reduce n by 1 and go to step 1
- Find the largest and put it in the end unless specified otherwise
- Best and worst case $O(n^2)$

Bubble Sort







- Ascending example
- Assuming you check if is sorted to improve time complexity
- For each outer loop iteration, i should get smaller, as the last item becomes the item before it each time
- In the inner loop from the start to i , swap two adjacent elements into a more ascending order
- Best case $O(n)$
- Worst case $O(n^2)$


Insertion Sort

- Descending order example
- 40, 13, 20, 8
- Keep first number same
- Compare 40 to 13, 13 is in correct place
- Compare 13 to 20, 20 is in wrong place, move 20 before 3
- 8 is in correct place
- Best case $O(n)$
- Worst case $O(n^2)$

Quicksort

- Recursive
- Done with the help of binary search
- Divide and conquer
- Divide
- Divide the large problem into smaller problems
- Recursively solve the smaller problems
- Conquer
- Combine the results of the smaller problems to produce result of the larger problem
- Recursion - base case

- When the recursion stops
- The algorithm
- Divide
- Choose an item p (pivot) and partition the item into 2 parts
- All items $< p$
- All items $\geq p$
- Recursively sort the two parts a) and b) mentioned above
- Conquer
- Do nothing
- Merge sort spends most of its time on conquer, and little time in divide, quicksort does the opposite
- 
- To partition $a[i \dots j]$ choose $a[i]$ as pivot p
- Remaining items (i.e. $a[i+1 \dots j]$) are divided into 3 regions
- $S1 = a[i+1 \dots m]$, items $< p$
- $S2 = a[m+1 \dots k-1]$, items $\geq p$
- Unprocessed = $a[k \dots j]$ are yet to be assigned to $S1$ or $S2$
- 
- m is an index border/separator of $S1$ and $S2$
- k is an index separator of $S2$ and unprocessed
- Initially both $S1$ and $S2$ are empty
- All items excluding p are in the unprocessed region
- For each items $a[k]$ in the unprocessed region
- Compare every item $a[k]$ with pivot p
- If items $a[k] \geq p$. Put $a[k]$ into $S2$
- Otherwise, put item $a[k]$ into $S1$
- Case 1: if item $a[k] \geq p$
- Go to next element by incrementing index k
- Case 2
- 
- 
- 
- Complexity
- Only a single for loop
- **of iterations = # of items in un-processed region = n**
- $n = (\text{high} - \text{low})$
- Complexity $O(n)$
- 

- Quicksort complexity
- Worst case analysis
- Each partition $O(n)$
- Worst case: n levels
- Complexity: $n+(n-1)+\dots+1$
- $= O(n^2)$
- Lower half only contains pivot
- Best case
- Always splitting array into two equal parts
- Depth of recursion: $\log n$
- Each level: n or fewer comparisons, so complexity $O(n \log n)$
- Worst case is rare
- In average we get some good splits and some bad ones
- Average complexity: $O(n \log n)$
- It is known that all comparison based sorting algorithms have a complexity lower bound: $n \log n$
- Hence, any comparison based sorting algorithm with worst-case complexity $= O(n \log n)$ is optimal
- Example
- 

Merge Sort

- Ascending order by default
- Two sorted halves
- 12,19,25,39
- 7,14,29
- Pointer at both first numbers
- Compare, copy the smaller one, and move the pointer in the corresponding list
- Steps
- Divide into two halves
- Recursively sort the halves
- Merge them
- Complexity
- Overall complexity: $O(n \log n)$

Heaps

- Uses trees
- The root and the external leaves
- Binary Trees

- 0, 1, or 2 children
- Add level by level, left to right
- Fill a level before moving to new level
- complete binary tree
- always filled nodes except for the leaves
- remaining unfilled leaf level should be as far left as possible
- if an array has blanks, it's not a complete binary tree
- Should be a complete binary tree
- max heap or min heap
- max heap
- data of the parent node \geq data of child node
- min heap
- data of the parent node is \leq data of child node
- static or dynamic
- static is easier to implement if index starting at one
- recap priority queues
- static
- length A is array length
- index node i
- parent i return $i/2$
- left i returns $2*i$
- right i returns $2*i+1$
- max heap property
- every heap is a complete binary tree
- but not every complete binary tree is not a heap
- Building
- start with nothing, build up the heap
- or convert complete binary tree to heap
- Max heapify
- assume binary trees rooted at LEFT(i)
- But A(i) may be smaller than its children
- violates max heap property
- Ex
- If left child larger than root
- Take a temporary index as index of the largest between the two
- Compare that with the right child
- Build max heap
- Start heapification from the parent of the last child
- Max heap insert

- always insert the element as the last element
- level by level, left to right
- heap size last index stored in i (it was increased by one)
- keep comparing last node with parent and swap if needed
- keep changing i to the parent's index
- Priority queue
- Lesser number is higher priority (higher rank)
- could be higher number for higher priority
- greater gpa has more priority
- xfinity example
- highest priority is the top of the queue
- if higher number is higher priority it is max heap
- if lower number is higher priority it is min heap
- start at root of the heap
- constant complexity
- this is the highest priority client
- delete the root and heapify
- priority queue: best to use heap for lowest time complexity
- start with max heap, you sort in ascending order as you delete the roots
- put the deleted root at the end of the array
- $n \log n$
- quick sort, heap sort, and merge sort
- deleting in binary tree
- swap first node with the last
- delete the last node
- then heapify again
- remove
- cheap if there is a heap
- max is set to highest element (root)
- swap last element with max (root)
- reduce max heap size by one
- max heapify (a,1)
- heapifying top down
- heap sort
- i is length of heap (for loop from i down to 2)
- swap first element with last element
- decrease heap size
- heapify at root
- examples

- check slides
- applications
- check slides
- in place sorting
- no quadratic worst-case
- selection algorithms
- graph algorithms
- ethernet
- dining philosopher's problem

Radix Sort

- Not in place
- Each item sorted as a character string
- No comparison of elements
- In each iteration
- Organize data into groups according to next character in each data
- Groups are then "concatenated" for the next iteration
- How many iterations?
- Max number of digits
- 10 groups
- Radix is base
- Dealing with decimal so its 10 groups
- How to sort
- Standardize the amount of digits by prepending zeros
- Start by looking at the smallest digit (rightmost digit)
- Assuming groups 0-9
- Put the number in its respective group based on its digit
- Rewrite the numbers as a list based on the order in the groups
- Group by group, value by value
- First iteration done
- Now, repeat the group sorting and rewriting
- Keep repeating up till you do the first digit and then rewrite it, then it will be sorted
- Algorithm
- Outer loop
- Look at each digit of the key
- Inner loop
- Take data from the array and put it in an array indexed by the digit
- 2nd loop
- Copies the data from the list back to the array

- Pseudocode
- Main program
- `int i`
- `int power = 1;` (power variable (10^0 , 10^1 , etc), initialized as 1)
- `int digitQueue[10];` (array of 10 queues)
- `for (i = 0; i < maxDigit; i++)`
- Distribute (for loop)
- Collect (for loop)
- `Power *= 10`
- `distribute(int v[], digitQ[], int power)`
- `int digit;`
- `for (int i = 0; i < v.size(); i++)`
- `digit = (v[i]/power) % 10`
- `digitQ[digit].enqueue(v[i]);` (array of queues)
- `collect(int digitQ[], int v[])`
- `int i = 0`
- `int digit`
- `for (digit = 0; digit < 10; digit++)`
- `while (!digitQ[digit] ...`
- Complexity
- $O(n)$ is the complexity for distribute and collect
- Number of iterations is d , the maximum number of digits amongst all the number...
- $O(dn)$ OR $O(n \log n)$ is the overall complexity
- But if you use d you have to define. explain it**