algo&comp

Friday, 28 April 2023 9:25 PM

1.
O() order of growth, increasing speed of growth
2021 + c and n0 of O()

2.
2017 - describe f(n), constant factors, lower-order terms, O()
2018 - Show quicksort (pivot)
2019 - O() of functions
2021 - O() of functions
2022 - write recursive function

3.
2017 - O() of functions
2018 - O() of functions
2019 - Linked lists: searching, worst case, deleting
2021 - Show quicksort (pivot)
2022 - Tree: Delete, insert

4.
2017 - Mergesort:
draw tree of recursive calls,
describe divide and conquer,
formulate recurrence relation,
asymptotic worst-case
2018 - Merge/insertion sort:
Show O(nk) worst case,
show sublist can merge in O(n log(n/k)
How to choose k
2019 - best worst ave of merge/insertion
Suggest List types where 1.)insertion faster than merge 2.)merge faster than insertion
2021 - BST: Define balance, purpose of balance, deleting, inserting
2022 - Binary min heap: heap property, tree representation,
Algorithm for : deleting minimum, inserting

5.
2017 - BST: inserting, deleting, suggest insertion to make BST not balanced
2018 - BST
2019 - BST: inserting, balance, non-balanced
2021 - AVL: insert and balance
2022 - Hashmap/treemap: advanatges, disadvantages

6.
2017 - Describe algorithm to solve problem and state running time O()
2018 - Valid heaps , draw trees
2019 - Worst cases given code: if using binary heap/ to insert/delete
2021 - Binary min heap: property, makeHeap() algorithm, draw tree,
2022 - Kruskal's algo, union find, draw min span tree, find no of MST

7.
2017 - Dijkstra's algo, draw shortest path tree, name faster algo, how to implement O(m log n)
2018 - Propose data structures
2019 - Prim's, Kruskal's, how to implement O(m log m)
2021 - Hash table: open addressing, linear probing
2022 - Draw recursion tree, O() after memoisation

8.
2017 - Describe algo for music files, state running time
2018 - Connected graph: DFS
2019 - Suggest data struc and state running times
2021 - Dijkstra's shortest path trees, O(m log n)
2022 - Suggest algo: select possible holiday dest given budget

9.
2018 - Suggest algo: plan travel connections between cities
2021 - outline algo to check plagiarism in essays. State O()

Block1

Big O Notation

f(n) = O(g(n)) as n-> infinity if and only if:
There exist constant c > 0, n0 >= 0 such that for every n > n0, we have f(n) <= c . g(n)

1 2 3 4 5 6 7 8 9

Question 3 Correct Mark 1.00 out of 1.00 Flag question

Sort the expressions below by increasing speed of growth.

$b^x = a \iff \log_b a = x$

1 < log n < n / (log n) < n < n.logn < n² < n¹⁰⁰⁰ < 2ⁿ < n!

SEARCHING

Linear search

```
boolean linearSearch(Array A, Element x)
{
  for (i=0 to A.size()-1) {
    if (A[i] == x)
      return true;
  }
  return false;
}
```

Worst: O(n)
Best: O(1)

Binary search Binary search

- *INPUT 要SORTED*
- *Always working with middle element*

Procedure binarySearch(Array A, Element x):
1. Let low, middle, high be the first, middle, last index of A
2. Let y=A[middle]
   2.1 If x==y, then x has been found – stop searching
   2.2 If x < y, look for x in the left half of A:
   recompute middle, high to point into left half; repeat 2
   2.3 If x > y, look for x in the right half of A:
   recompute low, middle to point into right half; repeat 2
3. Stop when indexes low, high cross

Worst case: O(log n)
Best case: O(1)

SORTING

Selection sort Selection sort

Loop whole array while updating smallest value

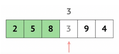
```
void selectionSort(Array A)
{
  for (i=0 to A.size()-1) {
    int smallest=i;
    for (j=i+1 to A.size()-1) {
      if (A[j] < A[smallest]) {
        smallest=j;
      }
      if (smallest != i) {
        (swap entries A[i], A[smallest]);
      }
    }
  }
```

$O(n^2) = n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$

```
for (j = 0; j < n-1; j++)
  int iMin = j;
  for (i = j+1; i < n; i++)
    if (a[i] < a[iMin])
      iMin = i;
  if (iMin != j)
    swap(a[j], a[iMin]);
```

Insertion sort Insertion sort

- *Work left to right*
- *Work whole array, compare to left element*
- *Insert item*




```
void insertionSort(Array A)
{
  for (i=1 to A.size()-1) {
    // Array A[0...i-1] already sorted.
    // Want to make A[0...i] sorted as well.
    int j=i;
    while ((j>0) && (A[j] < A[j-1])) {
      (swap entries A[j], A[j-1]);
      j=j-1;
    }
  }
}
```

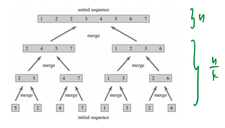
```
for i : 1 to length(A) -1
  j = i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j = j - 1
```

Worst case: O(n^2)
Best case O(n)

Merge sort Merge sort

- *Keep splitting into two*

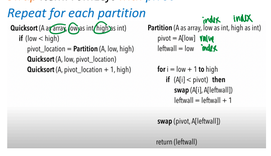
```
void mergesort(Array A, int low, int high)
{
  if (high <= low)
      return;
  int mid = (low+high)/2;
  mergesort(A, low, mid);
  mergesort(A, mid+1, high);
  merge(A, low, mid, high);
  // The merge call joins the sorted segments
  // A[low...mid] and A[mid+1...high] into a new
  // sorted list, then copies it into A[low...high]
}
```



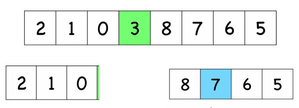
O(n log n)

Quicksort Quicksort

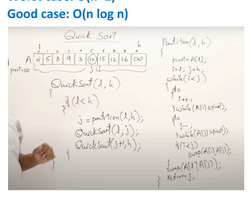
- *Pick pivot*
- *Move to right side*
- *From left, find first element > pivot. itemFromLeft*
- *From right, find first element < pivot. itemFromRight*
- *Swap*
- *Until -> index of itemFromLeft > index of itemFromRight*
- *Swap itemFromLeft with pivot*
- *Repeat for each partition*



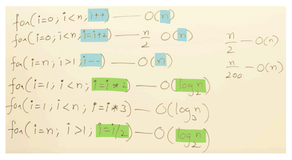
```
Quicksort (A as array, low as int, high as int)
  if (low < high)
    pivot_location = Partition (A, low, high)
    Quicksort (A, low, pivot_location)
    Quicksort (A, pivot_location + 1, high)

Partition (A as array, low as int, high as int)
  pivot = A[low]
  leftwall = low
  for i = low + 1 to high
    if (A[i] < pivot) then
      swap (A[i], A[leftwall])
      leftwall = leftwall + 1
  swap (pivot, A[leftwall])
  return (leftwall)
```



Worst case: O(n^2)
Good case: O(n log n)



Time complexity (for loop)



Time complexity

Printinorder lec08week18
Printpostorder lec08week18
Treenode respresentation lec08week18

ArrayList	**Linked lists**
Disadvantage: fixed size	**Disadvantage**:No random access(harder to navigate), uses more memory **Advantage**: easier to modify and extend **Constant time O(n)**

Assume the array/lists already contain n items:

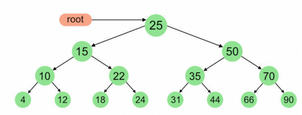
Operation	Time (array)	Time (linked list)
Add at 0	$\Theta(n)$	$\Theta(1)$
Delete 0	$\Theta(n)$	$\Theta(1)$
Add at end	$\Theta(1)$ unless full $\Theta(n)$ if full (rarely)	$\Theta(1)$
Delete from end	$\Theta(1)$	$\Theta(1)$
Random access	$\Theta(1)$	$\mathcal{O}(n)$
Delete in middle	$\Theta(n)$	$\Theta(1)$ with pointer $\Theta(n)$ without pointer

1. **Binary Search Tree**
O(log n)
Traversals
Insert according to 大小

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

OPS A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



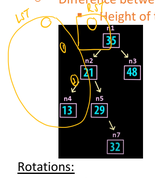

```
static BST insert(BST T, Key ik) {
  if (T == null)
    return new BST(ik);
  if (ik < T.key)
    T.left = insert(T.left, ik);
  else if (ik > T.key)
    T.right = insert(T.right, ik);
  return T;
}
```

Insertion
a. Left subtree < parent
b. Right subtree > parent

Deletion
a. 0 child
b. 1 child
Replace with child
c. 2 children: replace with
i. Largest elmnt of left subtree (Inorder predecessor) usually this
or
ii. Smallest elmnt of right subtree (Inorder seccessor)

IF ASK SEQUENCE JUST CONNECT LEFT RIGHT DOWN THE TREE, DON'T BUILD BST

AVL (self balancing)
Balanced
- Difference between height of left and right subtree for every node !> 1
  - Height of tree is number of edges of longest path between root and leaf



Height of LST = 2
Height of RST = 0
LST - RST = 2
>1, thus unbalanced

Rotations:
- LL: 转R
- RR: 转L
- LR: 转L，转R
- RL: 转R，转L

2. **Map ADT**
Store (key, value) pair
- **get(k)** - return value associated with k
- **put(k,v)**
- **remove(k)** - remove k and its value
- **containsKey(k)**
- **containsValue(v)**

HashSet	**HashMap**	**TreeMap**
- O(1) - For unordered sets and maps - Without duplicates - **insert(item)** - **contains(item)** - **delete(item)**	- **O(1): fast** - Not sorted - **Uses hashTable** - **get(k)** - **containsKey(k)** - **put(k,v)** - **remove(k)**	- **O(log n) (height of tree)** - Sorted - **Uses AVL**

3. **Hash tables**
- **O(1)**
- Slow
- Thread safe
- Size(number of contained elements)
- Capacity("space" reserved for further elements)

Hash functions time does not depend on number of keys
h(k) % m
- To store arbitrary objects
- String->hashcode->index

Collision handling(when two keys in the hash table share the same hash value):
a. **Separate Chaining**
b. **Open addressing- Linear probing**

4. **Priority queue**
- *Organise by priority*
- *Fast due to simple code*
- *Small memory usage (no extra space for "links"/ pointers or node objs)*

Min priority queue
- **insert(item)**
- **minItem()**
- **deleteMin()**

Max priority queue
- **insert(item)**
- **maxItem()**
- **deleteMax()**

As queue(linked list)	**As balanced trees**	**As Heaps (差不多跟tree一样)**
- **O(1)** - minItem(), deleteMin() - **O(n)** - insert(item) bc looking for right place to insert data	- **O(log n)** - insert, delete, minValue, delete	- **O(log n)** - insert, delete - **O(1)** - minValue - Ordered binary tree - **Max heap**: parent > children - **Min heap**: parent < children

Data structures wrap-up, pt. 2

- Binary Search Trees:
  - In "plain" version, not very good performance...
  - With self-balancing (rotations: AVL-trees or red-black trees), perform almost any reasonable operation in time $O(\log n)$
- Hash tables:
  - Set-operations $O(1)$ time, no support for min/max/successor
- Heaps: Basically, for implementing the Priority Queue ADT
  - Useful concept on its own (job scheduler, message queue, etc.)
  - Also useful inside other algorithms (e.g. A*-search: Keep P.Queue of most promising path candidates)
- Arrays: Good if data small or well-behaved, e.g., data arrives almost in order

5. **Binary Min Heap**
Top->bottom, left->right, all children >= parent

a. **Makeheap**
- *Create heap from an array*
- *Create max/min heap*

1. Determine the indices of its left and right child nodes.
2. Compare the value of the node with its child nodes and swap if necessary to satisfy the heap property.
3. Move up to the parent node and repeat steps 1 and 2 until the root node is reached.

b. **Heapsort**
- *Remove largest/ smallest item*
- *Call siftdown(uni)/siftup(uni)/heapify(online) to resort heap*

O(n log n) [n(numbers) ;log n(tree height)]

Insertion
- Add new element to end of heap
- Compare with parent. If < parent, swap
- Repeat

Deletion
- Remove root node and replace with last element
- Swap with smallest child node (even if two children > parent)
- Repeat

▶ Claim: To change the priority of the item in tree/array position i is easy:
1. increaseKey at position i: Change priority, call siftUp
2. decreaseKey at position i: Change priority, call siftDown

▶ Issue: We want to provide increaseKey(item, value), not increaseKey(position, value). Fixes?
- ▶ Suggestion: Augment with map: Map(Items, Positions)
- ▶ Update your map for every modification to the array (e.g., every swap)

▶ Code: siftUp(i)
1. While `array[i] < array[parent(i)]`:
   1.1 Swap contents `array[i]` and `array[parent(i)]`
   1.2 Let `i=parent(i)` (break if trying to find parent of node 1)

▶ Faster version: `makeHeap` algorithm:
1. Start with `n` numbers in an array in arbitrary order
2. For `i=n-1` down to 0:
   2.1 Call `siftDown(i)`

▶ Code `siftDown(i)`:
1. Let `smallest` be smallest node of `i`, `leftChild(i)`, `rightChild(i)`
2. If `smallest` $\neq$ `i`:
   2.1 Swap contents `array[i]` and `array[smallest]`
   2.2 Call `siftDown(smallest)` recursively

▶ Code for `heapsort(int[] array)`, array length n:
1. Call `makeHeap` to turn `array` into heap (in-place).
2. For `i = n-1` down to 0:
   2.1 Swap `array[0]` with `array[i]`
   2.2 "Forget" item `array[i]` in heap (decrease "size" parameter)
   2.3 Call `siftDown(0)` to put "new root" in place

Connected graph
- contains paths between every pair of vertices; starting from any vertex, we can traverse the graph to reach any other vertex through a sequence of edges.
- Formally, a graph G=(V,E) is connected if for every pair u,v that is an element of V, there exists a path in G between u and v.

Connectivity components
- Maximal connected subgraphs of a graph.
- Each connectivity component represents a set of vertices that are all reachable from each other, but not from any vertex in another component.
- *It is not possible to add any more vertices or edges to the subgraph without disconnecting it from the rest of the graph. I have no clue what this means ngl*

Breadth First Search(BFS)
Queue
Shortest paths BFS DFS
- Once explored new node, don't go down further, explore adjacent nodes
- No back edges (the new edge uv connects a node to its ancestor)
- Good for connectivity testing

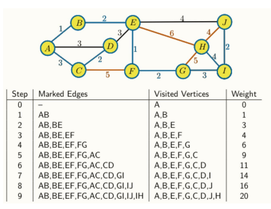
Depth First Search (DFS)
Stack
- Once explored new node, explore further down itself
- Revisit parent node, then visit adjacents
- No cross edges (any other case)
- Encode shortest paths

Spanning tree
Tree that connects every vertex
Minimum spanning tree
Spanning trees of minimum total cost

Prim's Algorithm

Prim's Algorithm from Lecture

- Data structures:
  1. Vertices marked/unmarked, initially all unmarked
  2. Priority Queue pq of edges, initially empty
  3. The tree T
- Use a "visit" subroutine process(v):
  1. Mark v as visited
  2. For every unmarked neighbour u of v:
     2.1 Add edge uv to pq with weight $w(uv)$
- Code, main loop:
  1. Select initial vertex v, process(v)
  2. Until pq is empty: // or "Until T has $|V|-1$ edges"
     2.1 e=pq.deleteMin() // pull cheapest edge e from pq
     2.2 If e has an unmarked endpoint u:
     add e to T, process(u)



Step	Marked Edges	Visited Vertices	Weight
0	–	A	0
1	AB	A,B	1
2	AB,BE	A,B,E	3
3	AB,BE,EF	A,B,E,F	4
4	AB,BE,EF,FG	A,B,E,F,G	6
5	AB,BE,EF,FG,AC	A,B,E,F,G,C	9
6	AB,BE,EF,FG,AC,CD	A,B,E,F,G,C,D	11
7	AB,BE,EF,FG,AC,CD,GI	A,B,E,F,G,C,D,I	14
8	AB,BE,EF,FG,AC,CD,GI,IJ	A,B,E,F,G,C,D,J	16
9	AB,BE,EF,FG,AC,CD,GI,IJ,IH	A,B,E,F,G,C,D,J,H	20

Kruskal's Algorithm Kruskal's algo and teaches when union()/find() is applied

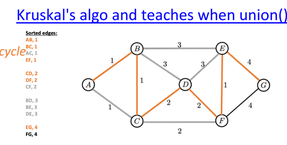
Kruskal's Algorithm from Lecture

To check if nodes belong to same group, so we don't create a cycle

We need a data structure for connected components:
- Find(v) – return a name for the connected component of T that contains v
- Union(u,v) – "merge" the components of u and v into one (by adding uv to T) *Merge groups tgt*

Efficient implementation using this:
1. Sort the edges E (e.g., merge sort) at $O(|E| \log |E|)$ time
2. Initiate your Union-Find structure to "empty"
3. For every edge uv in sorted order:
   3.1 If Find(u) == Find(v): ignore edge
   3.2 Otherwise call Union(u,v) and add to T

With a good Union-Find, the time is dominated by $O(|E| \log |E|)$.



Edges added:

None	{A}	{B}	{C}	{D}	{E}	{F}	{G}
AB	{A, B}		{C}	{D}	{E}	{F}	{G}
BC	{A, B, C}			{D}	{E}	{F}	{G}
AC	{A, B, C}			{D}	{E}	{F}	{G}
EF	{A, B, C}			{D}	{E, F}		{G}
CD	{A, B, C, D}				{E, F}		{G}
DF	{A, B, C, D, E, F}						{G}
CF	{A, B, C, D, E, F}						{G}
BD	{A, B, C, D, E, F}						{G}
BE	{A, B, C, D, E, F}						{G}
DE	{A, B, C, D, E, F}						{G}
EG	{A, B, C, D, E, F, G}						
FG	{A, B, C, D, E, F, G}						

Shortest paths
Dijkstra's Algorithm
Priority queue