Big-O: Upper bound of the complexity; its complexity is greater than the complexity of f(x) for all x greater than a constant c.

Big-omega: Lower bound of the complexity; its complexity is lower than the complexity of f(x) for all x greater than a constant c.

Big-theta: When Big-O and Big-omega are equal, Big-Theta is used to describe the complexity of the function.

MASTER THEOREM:

A: Number of subproblems looked at

B: Number of divisions made to the problem each recursive call.

C: Polynomial complexity of the nonrecursive code.

S > C:

C > S:

C == S:

SORTING (fuck):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 15 | 7 | 99 | 102 | 16 |

Selection (complexity: N2):

Iterate through the array, for each item, iterate through the array and swap current item with the min.

For(I = 0; I < length; i++)

{

Min = i;

For(J = I; J < length; J++)

{

If(arr[j] < arr[min])

Min = J

}

Swap(arr, I, min);

}

Insertion (complexity n2):

Starts from index 1, pushes each value as far left as it can go.

for(i = 1; i < n; i++)

for(j = i; j > 0 && arr[j].compareTo(arr[j-1]) < 0; j--)

swap(arr, j, j - 1);

Shell Sort (complexity nwhatthefuck)

Modified Insertion sort, uses intervals that shrink each time.

//Longer implementation:

Void privateInsert(int[] arr, int size, int start, int increment)

{

For(I = start+increment; I < size; I += increment)

For(j = I; I > start; j -= increment)

If(arr[j – increment] > arr[j])

Swap(arr, j – increment, j);

}

Void shell(int[] arr, int size)

{

Int incr, start;

Int THRESHOLD = 2;

For(incr = size/2; incr > THRESHOLD; incr = incr/2 + 1)

For(start = 0; start < incr; start++)

privateInsert(arr, size, start, incr=);

privateInsert(arr, size, 0, 1);

}

//Shorter Implementation:

void shellsort(int v[], int n)

{

int gap, i, j, temp;

for (gap = n/2; gap > 0; gap /= 2)

for (i = gap; i < n; i++)

for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {

temp = v[j];

v[j] = v[j+gap];

v[j+gap] = temp;

}

}

Radix exists. Idk.

Sorts by digit; works with integers.

Merge sort (complexity nlogn):

Important: use in any algorithm challenged to run in n log n complexity.

Trees:

Start from a root, have nodes coming off of branches; nodes with no branches coming out of them are called leaves.

Huffman Tree:

Compression where most used characters are the highest. Use bitstring to read.

Binary Tree

Null, non null node with two binary trees as children.

All nodes have 0, 1, or 2 children.

Full Binary Tree: All nodes have 0 or 2 children.

Expression Tree: FBT, internal nodes are operators, leaves are operands.

Complete Tree: Completely filled to level height – 2, last level filled left to right.

SBBST:

AVL:

Every node has order = height rc – height lc.

Order has to be -1, 0, or 1. Otherwise there’s a problem.

Rotate Right:

Temp = sr.lc

Sr.lc = temp.rc;

Temp.rc = sr;

Sr = temp;

Rotate Left:

Temp = sr.rc;

Sr.rc = temp.lc;

Temp.lc = sr;

Sr = temp;

Left Left Problem: Rotate Right on problem node.

Right Right Problem: Rotate Left on problem node.

Left Right Problem: Rotate Left on problem.lc, then rotate right on problem

Right Left Problem: Rotate Right on problem.rc, then rotate left on problem

Red-Black:

Nodes are red or black. Root is always black. Children of red must be black. When adding a node it starts as red. Number of black nodes from root to any child is the same. No two reds in a row.

Hashing needs further detail.

Open: separate list from hash table.

Closed; limited to hash table.

Hash function h(k), and Probe function p(k, i);

Example: h(k) = k % 10;

P(k, i) = (k + (i+1)) % capacityOfArray);

Capacity = 10;

Try to add k = 15:

H(15) = 5;

Put 15 into index 5 of array.

Try to add k = 25:

H(25) = 5

See index 5 is full in array, send 25 to index p(5, 0);

P(5, 0):

(5 + 0 + 1) % 10 = 6.

6 is empty, 25 goes to index 6.

Try to add k = 35:

H(35) = 5;

Index 5 is full in array, send 35 to p(5, 0);

P(5, 0)

(5 + 0 + 1) % 10 = 6

Index if full, call p(5, 1)

P(5, 1)

(5 + 1 + 1) % 10 = 7

7 is empty, 35 goes to index 7.

Removing: place a TOMBSTONE value in, and treat these values as full when adding. This allows for proper searching.

Graphs:

Collection of vertices and edges.

Two vertices connected by a single edge are adjacent.

Vertices have a degree: number of edges that go to it (or are just connected if undirected)

DAG: Directed Acyclic Graph. Used for networks and such.

Path: series of consecutive connected nodes (degrees?)

Simple path: path with no duplicates (acyclic)

Cycle: simple path except src = dst

Tour: every vertex is visited.

Subgraph: any collection of vertices in a graph.

Maximal Connected graph: all connected components.

Complete graph: edges between all pairs of vertices.

Clique: a complete subgraph.

Cyclic/Acyclic

Directed/Undirected

Digraph. Directed graph; edges have direction, point from source to destination.

Weighted/Unweighted

Connected/Disconnected

Complete/Incomplete