|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **BubbleSort**  void bubbleSort2(std::vector<int>& a){ bool swapped = true;  while(swapped){ swapped = false; for(size\_t i = 1; i < a.size(); ++i){ if(a[i - 1] > a[i]){ swap(a, i-1, i); swapped = true; } }}}  void bubbleSort3(std::vector<int>& a){ size\_t n, newn; n = a.size(); while(n > 0){ newn = 0; for(size\_t i = 1; i < n; ++i){ if(a[i - 1] > a[i]){ swap(a, i-1, i); newn = i;  } } n = newn; }}  Worst Case Time Complexity: O(n^2)Best Case Time Complexity: O(n)Average Case Time Complexity: O(n^2)  Worst Case Space Complexity: O(1) auxiliary  **Insertion Sort**  void insertionSort(int \* array, int len){ for(int j = 1; j < len; ++j){ int key = array[j]; int i = j - 1; //insert key into sorted sequence while((i >= 0) && (array[i] > key))  array[i + 1] = array[i]; --i; } array[i + 1] = key; }}  **Selecion Sort**  void selectionSort(std::vector<int>& a){ size\_t j, least;  for(size\_t i = 0; i < (a.size() - 1); ++i){ for(j = i + 1, least = i; j < a.size(); ++j){ if(a[j] < a[least]){ least = j;  } } swap(a, least, i); }}  ---------------------------------------------------------------------------  **Shell Sort**  void insertionSort(int inc, int start, std::vector<int>& data){  for(int key\_index = start; key\_index < data.size(); key\_index += inc){ int key = data[key\_index]; int other\_index = key\_index - inc; while(other\_index >= 0 && data[other\_index] > key){ data[other\_index + inc] = data[other\_index]; other\_index -= inc;  } data[other\_index + inc] = key; }}  void shellSort(std::vector<int>& data){ std::vector<int> increments; //produce the h's for(int h = 1; h < data.size(); ){ increments.push\_back(h); h = (3 \* h) + 1; } //loop over the different h's for(int i = increments.size() - 1; i >= 0; --i){ int h = increments[i]; //loop over the number of sub-arrays produced by the current h for(int start = 0; start <= h; ++start){ insertionSort(h, start, data); } }}  **Merge Sort**  int \* merge(int \* left, int \* right, int len){ int \* ret = new int[len + len]; int left\_position = 0; int right\_position = 0;  int ret\_position = 0; while(left\_position < len && right\_position < len){ int left\_value = left[left\_position];  int right\_value = right[right\_position]; if(left\_value < right\_value){ ret[ret\_position] = left\_value;  ret\_position++; left\_position++; } else {  ret[ret\_position] = right\_value; ret\_position++;  right\_position++; } }  while(left\_position < len){ ret[ret\_position] = left[left\_position]; ret\_position++; left\_position++;  } while(right\_position < len){ ret[ret\_position] = right[right\_position]; ret\_position++;  right\_position++; } return ret;}  int \* mergeSort(int \* input, int len){ if(len == 1){  return input; } int middle = len / 2; int \* left = new int[middle]; int \* right = new int[middle]; for(int i = 0; i < middle; ++i){ left[i] = input[i]; } for(int i = 0; i < middle; ++i){ right[i] = input[i+middle]; }  left = mergeSort(left, middle); right = mergeSort(right, middle); int \* ret = merge(left, right, middle);  delete [] left; delete [] right; return ret;}  ---------------------------------------------------------------------------  **Heap Sort**  algorithm fixDown(array d, int k, int size)  while leftChild(k) <= sizej = leftChild(k);  if there is a right child and the right child is larger  j = rightChild(k)if d[k] is less than d[j]  exchange(d[k], d[j])k = j;  algorithm createHeap(array d)  for i = d.length to 0  fixDown(d, i, d.length)  algorithm heapSort(array d)  createHeap(d);  for i = d.length to 1  exchange(d[0], d[i]);  fixDown(d, 0, i);  ---------------------------------------------------------------------------  **Bubble Sort kth min**  int select(std::vector<int>& data, int k){ for(int i = 0; i <= k; ++i){ int min\_index = i; int min\_value = data[i];  for(int j = i + 1; j < data.size(); ++j){  if(data[j] < min\_value){ min\_index = j;  min\_value = data[j]; } } exch(data, i, min\_index);  } return data[k];}  -----------------------------------------------------------------------------  **Quick Sort Kth min**  int partition(std::vector<int>& data, int left, int right, int pivot\_index){  int bound = data[pivot\_index];  exch(data, pivot\_index, right);  int store\_index = left;  for(int i = left; i <= right; ++i){  if(data[i] < bound && store\_index != i){  exch(data, store\_index, i);  store\_index++;  }  }  exch(data, right, store\_index);  return store\_index;  }  int select(std::vector<int>& data, int left, int right, int k){  if(left == right){  //if the list contains only one element, return it.  return data[left];  }  //select pivot index between left and right  int pivot\_index = (left + right) / 2;  int pivot\_new\_index = partition(data, left, right, pivot\_index);  int pivot\_dist = pivot\_new\_index - left;  //the pivot is in its final sorted position, so pivot\_dist reflects its 1-based position if data were sorted  if(pivot\_dist == k){  return data[pivot\_new\_index];  } else if(k < pivot\_dist){  return select(data, left, pivot\_new\_index - 1, k);  } else {  return select(data, pivot\_new\_index + 1, right, k - pivot\_dist - 1;  }  }  -----------------------------------------------------------------------------  **Radix Sort**  void radixSort(std::vector<long>& data){  const int radix = 10;  const int digits = 10; //(max number of digits for a long)  std::queue<long> queues[radix];  for(int i = 0, factor = 1; i < digits; factor \*= radix, ++i){  //fill up the queues  for(int j = 0; j < data.size(); ++j){  queues[(data[j] / factor) % radix].push(data[j]);  }  //empty the queues  int k = 0;  for(int j = 0; j < radix; ++j){  while(!queues[j].empty()){  data[k] = queues[j].front();  queues[j].pop();  ++k; } } }}  -----------------------------------------------------------------------------  **Counting Sort**  void countingSort(std::vector<int>& data){  const int max\_value = 10;  int counts[max\_value];  //clear counts  for(int i = 0; i < max\_value; ++i){  counts[i] = 0;  }  //calculate count of each number  for(size\_t i = 0; i < data.size(); ++i){  ++counts[data[i]];  }  //write back to the array  data.clear();  for(int i = 0; i < max\_value; ++i){  int c = counts[i];  for(int j = 0; j < c; ++j){  data.push\_back(i);  } }}  -----------------------------------------------------------------------------  **Stable Sort :** Bubble Sort, Insertion Sort, Merge Sort, Counting Sort, Radix Sort, Odd-Even Sort (at the end of the semester)  **Unstable Sort :** Selection Sort, Shell ort, Heap Sort, Smooth Sort, Quick Sort  -----------------------------------------------------------------------------  **External Sorting**  N/M = 15, you can see that there are 15 groups to merge to start out with  Then groups of 3 are merged together using 3 tapes and you get 15 / 3 = 5  5 is the number of groups that are input to the next pass  **Replacement Selection**  A priority queue of size M is used.  1. Initialization [step](http://trifort.org/ads/index.php/lecture/index/07/): M records are read into the priority queue (using, say, a min-heap)  2. Replacement [step](http://trifort.org/ads/index.php/lecture/index/07/): (creates a single run)  3. Remove the smallest element from the priority queue and write out  4. Read an element from the input tape. If the read element is smaller than the last output element, mark it as greater than all elements in the priority queue.  5.Stop a run when a marked element reaches the top of the queue  -----------------------------------------------------------------------------  **DSW rotation**  void rotateRight(TreeNode \* gr, TreeNode \* par, TreeNode \* ch){  //grandparent child becomes ch  if(gr->getRight() == par){ gr->setRight(ch); } else {  gr->setLeft(ch); } TreeNode \* ch\_right = ch->getRight();  par->setLeft(ch\_right); ch->setRight(par);}  void rotateLeft(TreeNode \* gr, TreeNode \* par, TreeNode \* ch){ //grandparent child becomes ch  if(gr->getRight() == par){ gr->setRight(ch);  } else { gr->setLeft(ch); }  TreeNode \* ch\_left = ch->getLeft();  par->setRight(ch\_left); ch->setLeft(par);}  **-----------------------------------------------------------------------------**  **DSW Algorithm**  algorithm createBackbone(root)  tmp = root;  while(tmp != NULL)  if tmp has a left child  rotate right about tmp and it's left child;  set tmp to the left child that just became the parent;  else  set tmp to the right child  algorithm createPerfectTree(n)  make n/2 left rotations starting from the top;  while (n > 1)  n = n / 2  make n left rotations starting from the top;  **-----------------------------------------------------------------------------**  **B-Tree Search Algo**  BTreeNode \* btreeSearch(int key, BTreeNode \* curr){  if(curr != NULL){  int i;  //seek the the closest key  for(i = 0; i < curr->sizeKeys() && curr->getKey(i) < key; ++i);  if(i >= curr->sizeKeys() || curr->getKey(i) > key){  //if the above for loop broke because the key was not found, recurse  return btreeSearch(key, curr->getPointer(i));  } else {  //else we found the node  return curr;  } } else { return NULL; }}  -----------------------------------------------------------------------------  **BTree Insert**  algorithm btreeInsert(root, key)  find a leaf node to insert key;  while(true)  find a proper position in array for key;  if node is not full  insert key in and shift other elements over;  return;  else  split node into node1 and node2;  distribute keys and pointers evenly between node1 and node2;  if node was the root  create a new root as parent of node1 and node2;  put the middle key from the split node in the root and set the pointers;  return;  else  node = node's parent;  -----------------------------------------------------------------------------  **BTree Delete**  1: algorithm btreeDelete(root, key)  2: node = BTreeSearch(key, root);  3: if(node == null)  4: return;  5: if(node is not a leaf)  6: find a leaf with the closest predecessor S of key;  7: copy S over key in node;  8: node = the leaf containing S;  9: remove S from node;  10: else  11: delete key from node;  12: while(true)  13: if node does not underflow  14: return;  15: else if there is a sibling of node with enough keys  16: redistribute keys between node and its sibling;  17: return;  18: else if node's parent is the root  19: if the parent has only one key  20: merge node, its sibling, and the parent to form a new root;  21: else  22: merge node and its sibling;  23: return;  24: else  25: merge node and its sibling;  26: node = its parent;  **-----------------------------------------------------------------------------**  **Digital Search Tree**  algorithm binaryTrieInsertion(Node root, long key){  2: int digit = 0;  3: Node curr = root;  4: Node prev = NULL;  5: 6: //handle root case  7: 8: while(true){  9: if(curr is null){  10: curr = new Node(key);  11: make curr have a value;  12: make correct prev left/right pointer point to curr;  13: return;  14: }  15: if(curr is a leaf){  16: Node temp = new Node(key);  17: make temp have a value;  18: Node new\_node = split(temp, curr, digit);  19: make correct prev left/right pointer point to new\_node;  20: }  21: prev = curr;  22: curr = curr->left if value of key at digit is 0, otherwise it is equal to curr->right;  23: ++digit;  24: }  25: }  26:  27: algorithm split(Node left, Node right, int digit)  28: {29: Node new\_node = new Node();  30: if(left->value and right-value are different at digit){  31: setup new\_node->left and new\_node->right to point (correctly) to left and right;  32: } else {  33: if(both left->value and right->value are 0 at digit){  34: new\_node->setLeft(split(left, right, digit+1);  35: } else {  36: new\_node->setRight(split(left, right, digit+1);  37: }  38: }  39: } | Hilbert Pack Algorithm for Bulk Insertion  1. Assign the hilbert value to each rectangle  2. Sort rectangles according to hilbert value  3. Create leaf nodes: Evenly divide rectangles based on hilbert value  4. Create inner nodes: Recursively build the tree from the leaf nodes using the extents of the children. The multiple children of one parent node are chosen using the time-based ordering of creation. The time based ordering is just the current value of an integer counter.  -------------------------------------------------------------------------------  Hilbert space filling curve example  For all points, find max\_x and max\_y.  1. Find the max of max\_x and max\_y and call this max\_xy.  2. If max\_xy is a [power](http://trifort.org/ads/index.php/lecture/index/11/) of two, leave it. Else make it the next higher power of two  For each point:  1. Initialize w to be max\_xy / 2. Dist = 0.  2. Find the quadrant on a hilbert curve that the point is in.  3. Dist += (quadrant \* w \* w)  4. Calculate xnew and ynew according to the formulas  5. w becomes w / 2  6. Repeat steps 2 to 5 until w becomes 0   |  |  |  | | --- | --- | --- | | Quadrant | x\_new | y\_new | | 0 | Y | X | | 1 | X | y - w | | 2 | x - w | y - w | | 3 | w - y - 1 | w \* 2 - x - 1 |   --------------------------------------------------------------------------------------  Example with coordinates: (4, 0): sfc Quadrant = 3. Dist = 3 \* w \* w = 3 \* 4 \* 4  [Apply](http://trifort.org/ads/index.php/lecture/index/11/) table: x,y=(3, 3)  Quadrant = 2. Dist += 2 \* w \* w = 2 \* 2 \* 2  [Apply](http://trifort.org/ads/index.php/lecture/index/11/) table: x,y=(1, 1)  Quadrant = 2. Dist += 2 \* w \* w = 2 \* 1 \* 1  Dist = 58  --------------------------------------------------------------------------------------  KD Tree  kd-tree mem  void insert(Point p, TreeNode \* curr, bool is\_x){  if(is\_x){  if(p.x < curr->getX()){  follow\_left\_node(!is\_x);  } else {  follow\_right\_node(!is\_x);  }  } else {  if(p.y < curr->getY()){  follow\_left\_node(!is\_x);  } else {  follow\_right\_node(!is\_x);  }  }  }  --------------------------------------------------------------------------------------**Octree**  void insert(Point p, Octree \* curr){  int quad = findQuad(p, curr);  if(curr has no child at quad){  insert child into curr->children[quad];  } else {  insert(p, curr->children[quad]);  }  }  --------------------------------------------------------------------------------------  **Interval Trees**  1. Each node has a low\_endpoint field  2. Each node has a high\_endpoint field  3. Each node has the max of the high\_endpoints of itself and its children  4. Each node has a left subtree where all intervals have low\_endpoints <= curr\_low\_endpoint  5. Each node has a right subtree where all intervals have low\_endpoints > curr\_low\_endpoint  --------------------------------------------------------------------------------------**Kth-Min Tree**  [Start](http://trifort.org/ads/index.php/lecture/index/11/): curr = root. Looking for k=5.  KthMinTreecount\_left = countNodes(curr->getLeft()) = 6; k < count\_left. findKth(curr->getLeft(), k).  KthMinTreecount\_left = 2. k > count\_left. findKth(curr->getRight(), k - count\_left - 1) KthMinTreecount\_left = 1. k = 2. k > count\_left. findKth(curr->getRight(), k - count\_left - 1)KthMinTreecount\_left = 0. k = 0. k == count\_left, return curr value  --------------------------------------------------------------------------------------  **Graphs Depth First Search**  1: algorithm depthFirstSearch()  2: for all vertices u  3: visited(u) = false;  4: while there is a vertex v such that visited(v) == false  5: searchNode(v);  6:  7: algorithm searchNode(Node v)  8: visit node  9: for all adjacent vertices u  10: searchNode(u)  --------------------------------------------------------------------------------------**Graphs Breadth First Search**  1: algorithm breadthFirstSearch()  2: for all vertices u  3: visited(u) = false;  4: edges = null  5: while there is a vertex v such that visited(v) == false  6: enqueue(v);  7: while queue is not empty  8: v = dequeue();  9: for all vertices u adjacent to v  10: if visited(u) is false  11: visited(u) = true;  12: enqueue(u);  13: attach edge(vu) to edges;  14: output edges;  --------------------------------------------------------------------------------------  **Dijkstra's Algorithm is one of the first label-setting methods.**  1: algorithm dijkstra(weighted simple digraph, vertex first)  2: for all vertices v  3: currDist(v) = infinity;  4: currDist(first) = 0;  5: toBeChecked = all vertices;  6: while toBeChecked is not empty  7: v = a vertex in toBeChecked with minimal currDist(v);  8: remove v from toBeChecked;  9: for all vertices u adjacent to v and in toBeChecked  10: if currDist(u) > currDist(v) + weight(edge(vu))  11: currDist(u) = currDist(v) + weight(edge(vu));  12: predecessor(u) = v;  **--------------------------------------------------------------------------------------**  **Dikstra code**  void dij(int n,int v,int cost[10][10],int dist[])  {   int i,u,count,w,flag[10],min;   for(i=1;i<=n;i++)    flag[i]=0,dist[i]=cost[v][i];   count=2;   while(count<=n)   {    min=99;    for(w=1;w<=n;w++)     if(dist[w]<min && !flag[w])      min=dist[w],u=w;    flag[u]=1;    count++;    for(w=1;w<=n;w++)     if((dist[u]+cost[u][w]<dist[w]) && !flag[w])      dist[w]=dist[u]+cost[u][w];   } }  **--------------------------------------------------------------------------------------**  **Bellman Ford Algo**  1: algorithm bellmanFord(weighted simple digraph, vertex first)  2: for all vertices v  3: currDist(v) = infinity;  4: currDist(first) = 0;  5: while there is an edge(vu) such that currDist(u) > currDist(v) + weight(edge(vu))  6: currDist(u) = currDist(v) + weight(edge(vu));  --------------------------------------------------------------------------------------  Warshall Floyd  1: algorithm warshalFloydIngerman(adjacency matrix weight)  2: for i = 1 to |V|  3: for j = 1 to |V|  4: for k = 1 to |V|  5: if weight[j][k] > weight[j][i] + weight[i][k]  6: weight[j][k] = weight[j][i] + weight[i][k];  --------------------------------------------------------------------------------------  **Undirected graphs cycle detection**  1: bool cycleDetect(IVertex \* v)  2: vector<IVertex \*> path;  3: set<IVertex \*> visited;  4: add v to path;  5: add v to visited;  6: return doCycleDetect(v, path, visited);  7:  8: bool doCycleDetect(IVertex \* v, vector<IVertex \*>& path, set<IVertex \*>& visited)  9: foreach(edge 'e' in v->getEdges)  10: IVertex \* target = e->getTarget();  11: if( last element in path == target)  12: continue;  13: if( visited contains target )  14: return true;  15: add v to path;  16: add target to visited;  17: if(doCycleDetect(target, path, visited))  18: return true;  19: else  20: path.pop\_back();  21: return false;  --------------------------------------------------------------------------------------**Directed graph cycle detection**  1: algorithm digraphCycleDetectionDFS(vertex v)  2: num(v) = i++;  3: for all vertices u adjacent to v  4: if num(u) is 0  5: digraphCycleDetectionDFS(u);  6: else if num(u) is not infinity  7: cycle detected;  8: num(v) = infinity;  --------------------------------------------------------------------------------------  **Minimum Spanning Tree**  **1**: algorithm kruskalAlgorithm(weight connected undirected graph)  2: tree = null;  3: edges = sequence of all edges of graph sorted by weight;  4: for(i = 1; i <= |E| and |tree| < |V| - 1; ++i)  5: if(e\_i from edges does not form a cycle with edges in tree  6: add e\_i to tree;  --------------------------------------------------------------------------------------  **Djikstra Algo spanning tree**  1: algorithm dijkstraSpanningTree(weight connected undirected graph)  2: tree = null;  3: edges = an unsorted sequence of all edges of graph;  4: for(j = 1 to |E|)  5: add e\_j to tree;  6: if(there is a cycle in tree)  7: remove an edge with maximum weight from this only cycle;  --------------------------------------------------------------------------------------  **Block DFS**  1: algorithm blockSearch()  2: for all vertices v  3: num(v) = 0;  4: i = 1;  5: while there is a vertex v such that num(v) == 0  6: blockDFS(v, NULL);  7:  8: algorithm blockDFS(vertex v, vertex parent\_v)  9: pred(v) = num(v) = i++;  10: for all vertices u adjacent to v  11: if edge(uv) has not been processed  12: push(edge(uv));  13: if num(u) is 0  14: blockDFS(u, v)  15: if pred(u) <= num(v)  16: e = pop();  17: while e != edge(vu)  18: output e;  19: e = pop();  20: output e;  21: else  22: pred(v) = min(pred(v), pred(u))  23: else if u is not parent\_v  24: pred(v) = min(pred(v), num(u))  --------------------------------------------------------------------------------------  **Strong DFS**  1: algorithm strongDFS(vertex v)  2: pred(v) = num(v) = i++;  3: push(v);  4: for all vertices u adjacent to v  5: if num(u) is 0  6: strongDFS(u);  7: pred(v) = min(pred(v), pred(u));  8: else if num(u) < num(v) and u is on stack  9: pred(v) = min(pred(v), num(u))  10: if pred(v) == num(v);  11: w = pop();  12: while w != v  13: output w;  14: w = pop(); | 15: output w;  16:  17: algorithm strongSearch()  18: for all vertices v  19: num(v) = 0;  20: i = 1;  21: while there is a vertex v such that num(v) == 0  22: strongDFS(v);  -----------------------------------------------------------------  **Topological Sort**  Summary algorithm:  1: algorithm topologicalSort(digraph g)  2: for i = 1 to |V|  3: find a minimal vertex v;  4: num(v) = i;  5: remove from g vertex v and all edges incident with v;  **Algorithm Based on DFS**:  1: algorithm TS(v)  2: num(v) = i++;  3: for all vertices u adjacent to v  4: if num(u) == 0  5: TS(u);  6: else if TSNum(u) == 0;  7: error a cycle was detected;  8: TSNum(v) = j++;  9:  10: algorithm topologicalSorting(digraph g)  11: for all vertices v  12: num(v) = TSNum(v) = 0;  13: i = j = 1;  14: while there is a vertex v such that num(v) == 0  15: TS(v);  16: output vertices according to their TSNum;  ------------------------------------------------------------------  Ford Fulkerson Algo  1: algorithm findPath(node source, node sink, path)  2: if(source == sink)  3: return path;  4: foreach edge in source.edges  5: residual = edge.capacity - edge.flow;  6: if residual > 0 and edge is not in path  7: new\_path = findPath(edge.target, sink, path + edge);  8: if new\_path is not EMPTY\_PATH  9: return new\_path;  10: foreach edge in edges pointing to source (reverse edge)  11: flow = forward edge flow  12: if flow > 0 and edge is not in path  13: new\_path = findPath(edge.target, sink, path + edge);  14: if new\_path is not EMPTY\_PATH  15: return new\_path;  16: print "backtracking at: " + source;  17: return EMPTY\_PATH;  18:  19: algorithm fordFulkerson(digraph g, node source, node sink)  20: path = findPath(g, source, sink, NULL);  21: printPath(path);  22: while path != EMPTY\_PATH  23: foward\_flow = min of all residuals for each forward edge in path  24: reverse\_flow = min of all forwards flows for each reverse edge in path  25: min\_flow = min(forward\_flow, reverse\_flow);  26:  27: //augment path  28: foreach edge in path  29: if edge is forward  30: edge.flow += min\_flow;  31: if edge is reverse  32: fedge = find foward edge for edge  33: fedge.flow -= min\_flow;  34: path = findPath(g, source, sink, NULL);  35: printPath(path);  36: return the sum of edge.flow forall the edges in source.edges  ------------------------------------------------------------------  Maximum Matching  1: algorithm findMaximumMatching(bipartite graph){  2: for(all unmatched vertices v){  3: set level of all vertices to 0;  4: set parent of all vertices to null;  5: level(v) = 1;  6: last = null;  7: clear queue;  8: enqueue(v);  9: while(queue is not empty and last is null){  10: v = dequeue();  11: if(level(v) is an odd number){  12: for(all vertices u adjacent to v such that level(u) is 0){  13: if(u is unmatched){  14: parent(u) = v;  15: last = u;  16: break;  17: } else if(u is matched but not with v){  18: parent(u) = v;  19: level(u) = level(v) + 1;  20: enqueue(u);  21: }  22: }  23: } else { //if level(v) is an even number  24: enqueue(vertex u matched with v);  25: parent(u) = v;  26: level(u) = level(v) + 1;  27: }  28: }  29: if(last is not null){ //augment matching by updating the augmenting path;  30: for(u = last; u is not null; u = parent(parent(u))){  31: matchedWith(u) = parent(u);  32: matchedWith(parent(u)) = u;  33: }  34: }  35: }  36: }  ------------------------------------------------------------------  **Hash Map Insertion Insertion**  1. Compute the index from hash(key)  2. Take a pointer to the head of the list  3. Make a new node in the linked list and set the next to the heap  4. Make the head point to the new node  Deletion  1. Compute the index from hash(key)  2. Search the linked list to find the prev and next of the curr  3. Use regular linked list removal  ------------------------------------------------------------------  **LZW algo**  1: compressing: aababacbaac  2: match: 0 for: a  3: adding symbol: aa at 26  4: match: 0 for: a  5: adding symbol: ab at 27  6: match: 1 for: b  7: adding symbol: ba at 28  8: match: 27 for: ab  9: adding symbol: aba at 29  10: match: 0 for: a  11: adding symbol: ac at 30  12: match: 2 for: c  13: adding symbol: cb at 31  14: match: 28 for: ba  15: adding symbol: baa at 32  16: match: 30 for: ac  17: adding symbol: ac at 33  -------------------------------------------------------------------------------------------------------  **Time Complexities**   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Sort | Description | In-Place | Worst Case Time Complexity | Sorted Time Complexity | Average Time Complexity | | Bubble Sort | After each outer loop, the ith ranked element is at position i. | Yes | O(n^2) | O(n) | O(n^2) | | Insertion Sort | Can insert a single key into a sorted sequence in O(n). | Yes | O(n^2) | O(n) | O(n^2) | | Selection Sort | Finds the ith ranked element and places it in position i. Used in parallel computing. | Yes | O(n^2) | O(n^2) | O(n^2) | | Shell Sort | Solves the rabbits and turtles problem for insertion sort and bubble sort. | Yes | Unknown | O(n) | Unknown | | Merge Sort | Breaks the array or list into two parts and uses divide and conquer. | No | O(nlgn) | O(nlgn) | O(nlgn) | | Heap Sort | Uses a max heap to sort. | Yes | O(nlgn) | O(nlgn) | O(nlgn) | | Smooth Sort | Uses leonardo heaps to sort. | Yes | O(nlgn) | O(n) | O(nlgn) | | Quick Sort | Recursively Divides the elements into smaller and greater than a key. Good with cache. | Yes | O(n^2) | O(n) | O(nlgn) | | Intro Sort | Uses quick sort until the depth of recursion goes beyond lgn. Then switches to in-place heapsort | Yes | O(nlgn) | O(n) | O(nlgn) | | Radix Sort | Uses modulus and division to sort (not comparisons). | No | O(dn) | O(dn) | O(dn) | | Counting Sort | A non-comparison sort that is good for small ranges of values. | No | O(n) | O(n) | O(n) |   ---------------------------------------------------------------------------------------------------  Kth Min   |  |  |  |  | | --- | --- | --- | --- | | Algorithm | Description | Worst Case Time Complexity | Average Time Complexity | | Sort | Sort then go to kth element. Good if you are finding the kth min many times | O(nlgn)-O(n) (Sort time) | O(nlgn)-O(n) (Sort time) | | Bubble Sort Kth Min | Do the first k outer steps of bubble sort | O(kn) | O(kn) | | Quick Sort Kth Min | Use the partitioning from quicksort | O(n^2) | O(n) | | Min Heap Kth Min | Given a min heap, run the first k steps of heapsort | O(klgn) | O(klgn) | | Kth Min Tree | Given a balanced kth min tree, execute the algorithm on the tree | O(lgn) | O(lgn) |   ---------------------------------------------------------------------------------------------------  Tree Vocabulary   |  |  | | --- | --- | | Word | Description | | DSW Algorithm | Stop the world to balance tree | | AVL Tree | O(lgn) insertion/deletion/remove | | Red-Black Tree | O(lgn) insertion/deletion/remove | | Splay Tree | O(lgn) insertion/deletion/remove (amortized) | | B-Tree | Good for databases | | R-Tree | Good for range queries | | 2-3-4 Tree | Every 2-3-4 tree has a corresponding red-black tree | | Binary Trie | O(lg(bits)) height with no rotations | | k-d Trees | Easily keep multi-dimensional data without R-Trees. |   ------------------------------------------------------------------------------------------------------------  Calculate Height  73: int calcHeight(TreeNode \* curr){  74: if(curr == NULL){  75: return 0;  76: }  77: int left\_height = calcHeight(curr->getLeft());  78: int right\_height = calcHeight(curr->getRight());  79: if(left\_height > right\_height){  80: return left\_height + 1;  81: } else {  82: return right\_height + 1;  83: }  84: }  ------------------------------------------------------------------------------------------------------------  **Time Complexities Graphs**  Dijkstra's  Using a simple data structure for currDist that requires O(|V|) to find the min:  The while at line 6 is executed O(|V|) times. Then for each of these executions, the search for min requires O(|V|) time.  The for loop at line 9 is executed O(|E|) times.  Total time: O(|V|^2+|E|)  Using a min-heap for currDist:  The outer while executes O(|V|) times and for each of those, removing from the min-heap takes O(lg|V|) time. This steptakes O(|V|\*lg|V|) time.  Updating currDist when a min distance is found, updating the key in the min-heap takes O(lg|V|) time. This is executed for every edge in the graph so this step takes O(|E|\*lg|V|) time  Total time: O((|V|+|E|)\*lg|V|)  The total number of edges in a complete graph: |E| = O(|V|^2)  In a complete graph, using a min-heap takes O((|V|+|V|^2)\*lg|V|), but with a sparse graph, using a min-heap is faster. In sparse graphs, dijkstra's algorithm with a min-heap is the fastest known single-source shortest-path algorithm for graphs with nonnegative weights. [1]  ------------------------------------------------------------------------------------------------------------  In the **Bellman-Ford algorithm**, the first pass finds all the one edge shortest paths, the second: two edge shortest paths, etc.  The time complexity is: O(|V|\*|E|). There will be at most |V|-1 passes through the sequence of |E| edges, because |V|-1 is the largest number of edges in any path.  ------------------------------------------------------------------------------------------------------------  The time complexity of the WFI-Algorithm is O(|V|^3). This is good for dense graphs, but for sparse graphs we can repeatedly use a single-source method on all the vertices to get something like O((|V|^2+|E||V|)\*lg|V|) (this uses dijkstra's algorithm with a heap).  **The WFI-Algorithm** can be used to detect cycles in a graph if the diagonal is initialized to infinity. If any of the diagonal values are changed, the graph contains a cycle.  ------------------------------------------------------------------------------------------------------------  Kruskal Algo  The time complexity of Kruskal's Algorithm is dependent on  The sorting method used: O(|E|lg|E|)  The for loop: O((|V|-1)|V|) = O(|V|^2)  The max number of edges in a graph is |V|^2, so the time complexity is O(|E|lg|E|)  **Dijkstar Spanning Tree**  Time complexity: O(|E||V|).  The for loop executes |E| times  Each time checking for a cycle takes |V| time  ------------------------------------------------------------------------------------------------------------  **Block Search**  Time complexity of blockSearch: O(|V| + |E|), Strong search same  ------------------------------------------------------------------------------------------------------------  **Undirected Graph cycle detection**  A simple modification to depth-first search can be done for undirected graphs that takes O(|V| + |E|) time.  -------------------------------------------------------------------------------------------------------- | **Maximum Flows**  The algorithm is never guarenteed to terminate, but if it does:  O(Ef) - E is the number of edges, f is the max flow value  Each edge gets incremented a flow value by at least 1 each time until the max is reached  ------------------------------------------------------------------------------  **Kcore clustering**  Time complexity:  O(m): cores decomposition  O(m \* max(deg, lgn)): finding cores  m: number of edges.  deg: maximum degree.  n: number of vertices.  --------------------------------------------------------------------------  **Quick Sort:**  void quicksort (std :: vector <int> &vec, int first, int last){  int i=first, j=last;  int tmp;  int pivot = vec[abs((first + last)/2)];  while (i<=j)  {  while (vec[i] < pivot)  i++;  while (vec[j] > pivot)  j--;  if (i<=j)  { tmp=vec[i];  vec[i]=vec[j];  vec[j]=tmp;  i++;  j--;  }  }  if (first < j)  quicksort(vec,first,j);  if (i < last)  quicksort(vec,i,last);  }  --------------------------------------------------------------------------  **Running Median**  std::vector<int> RunningMedian :: compute(std::vector<int> vec, int window\_size)  {  std :: vector<int> dup = vec;  std :: vector < int > output;  double median;  int start = 0;  int end = 1;  int call\_quicksort =0;  for (int i=start; i<end; i++)  {  if (call\_quicksort)  quicksort (dup,start,end-1);  if ( i%2 == 1)  median = (dup [i/2] + dup [(i/2)+1])/2;  else  median = dup [i/2];  output.push\_back(median);  int traverse = end-1;  int flag =0;  while (traverse > start)  {  if (dup [end-1] < dup[traverse -1])  traverse --;  else  {  flag =1;  break;  }  }  if (dup [end -1] < dup [end -2])  call\_quicksort =1;  }  while (start < vec.size() - window\_size)  {  dup = vec;  start ++;  if (end < vec.size())  {  end ++;  }  quicksort (dup,start,end-1,0);  int i =(start+end)/2;  median = (dup [i] + dup [i-1])/2;  output.push\_back(median);  }  return(output);  }  --------------------------------------------------------------------------  **Smooth Sort**  void SmoothSort :: smoothsort (std::vector<double>& vec, unsigned int num){  int index;  int size=0;  for (int i =0; i < vec.size(); i++)  {  index = i+1;  int pos=0;  prev\_root.clear();  while (checkfibo(index) == false)  {  pos = pos + check\_near(index);  prev\_root.push\_back(pos - 1);  index = index - check\_near(index);  }  dup.push\_back(index);  prev\_root.push\_back(i);  copy\_vector(vec);  root\_sort(vec);  copy\_vector(vec);  **}**  void SmoothSort :: create\_heap (int i,std :: vector<double> &vec)  {int index = i;  double left;  double right;  double max;  double swap=vec[index];  int new\_index= index;  while (dup[index] > 1)  {  max = vec [index];  right = vec[index-1];  left = vec[(index-1) - dup[index-1]];  if (right < left)  {  swap = left;  new\_index = (index-1) - dup[index-1];  }  else  {  swap = right;  new\_index = index -1;  }  if (max > swap)  break;  double var = swap;  vec[new\_index] = vec [index];  vec[index] = var;  index = new\_index;  }  }  void SmoothSort :: copy\_vector (std::vector<double>& vec)  {for (int i =0; i< prev\_root.size(); i++)  {  create\_heap(prev\_root[i],vec);  }  }  void SmoothSort :: root\_sort (std::vector<double>& vec)  {for (int i=0; i<prev\_root.size() ; i++)  {  for (int j=i+1; j<prev\_root.size() ; j++)  {  if (vec[prev\_root[i]] > vec[prev\_root[j]])  {  double swap;  swap = vec[prev\_root[i]];  vec[prev\_root[i]] = vec[prev\_root[j]];  vec[prev\_root[j]] = swap;  }}}}  ---  void SmoothSort :: sortify(std::vector<double>& vec)  {int right, left;  int prev\_right = -1, prev\_left = -1;  int i=prev\_root.size() - 1;  right = prev\_root[i] - 1;  while (right != 1)  {if (dup[prev\_root[i]] != 1)  {  right = prev\_root[i] - 1;  left = right - dup[right];  prev\_root.pop\_back ();  prev\_root.push\_back(left);  prev\_root.push\_back(right);  }  else  prev\_root.pop\_back ();  root\_sort(vec);  copy\_vector(vec);  i=prev\_root.size() - 1;  }  --------------------------------------------------------------------------  **Merge Sort**  #include<stdio.h>  #include <stdlib.h>  struct node  {  int number;  struct node \*next;  };  struct node \*addnode(int number,struct node \*next);  struct node\*mergesort(struct node \*head);  struct node \*merge(struct node \*one,struct node \*two);  int main(void){  struct node \*head;  struct node \*current;  struct node \*next;  int test[]={8,3,1,4,2,5,7,0,11,14,6};  int n=sizeof(test)/sizeof(test[0]);  int i;  head=NULL;  for (i=0;i<n;i++)  head=addnode(test[i],head);  i=0;  head=mergesort(head);  printf("before----after sort \n");  for (current=head;current!=NULL;current=current->next)  printf("%4d\t%4d\n",test[i++],current->number);  /\* free list \*/  for (current=head;current!=NULL;current=current->next)  next=current->next;free(current);  return 0;  }  struct node \*addnode(int number,struct node\* next){  struct node \*tnode;  tnode=(struct node\*)malloc(sizeof(\*tnode));  if(tnode!=NULL){  tnode->number=number;  tnode->next=next;  }  return tnode;  }  struct node \*mergesort(struct node \*head){  struct node \*head\_one;  struct node \*head\_two;  if((head==NULL) ||(head->next==NULL))  return head;  head\_one=head;  head\_two=head->next;  while( (head\_two!=NULL) &&(head\_two->next!=NULL)){  head=head->next;  head\_two=head->next->next;  }  head\_two=head->next;  head->next=NULL;  return merge(mergesort(head\_one),mergesort(head\_two));  }  struct node \*merge(struct node\*head\_one,struct node\*head\_two){  struct node \*head\_three;  if(head\_one==NULL)  return head\_two;  if(head\_two==NULL)  return head\_one;  if(head\_one->number<head\_two->number){  head\_three=head\_one;  head\_three->next=merge(head\_one->next,head\_two);  }  else  {  head\_three=head\_two;  head\_three->next=merge(head\_one,head\_two->next);  }  return head\_three;  }  --------------------------------------------------------------------------  **AVL Tree Insert**  void AVLTree :: insert(int key)  {  AVLTreeNode \*temp\_ptr = new AVLTreeNode();  AVLTreeNode \*tr;  AVLTreeNode \*prev;  vector <AVLTreeNode \*> parent\_vec;  temp\_ptr->setKey(key);  temp\_ptr->setRight(NULL);  temp\_ptr->setLeft(NULL);  if (firsttime == 0)  {  firsttime = 1;  root = temp\_ptr;  }  else  {  tr = root;  prev = root;  while (tr != NULL)  {  if (key > tr->getKey())  {  parent\_vec.push\_back(tr);  prev=tr;  tr = tr->getRight();  }  if (tr == NULL)  {  break;  }  if (key < tr->getKey())  {  parent\_vec.push\_back(tr);  prev=tr;  tr = tr->getLeft();  }}  if (key > prev->getKey())  {  parent\_vec.push\_back(prev);  prev->setRight(temp\_ptr);  }  else  {  prev->setLeft(temp\_ptr);  parent\_vec.push\_back(prev);  }}  AVLTreeNode \*node;  int left\_height;  int right\_height;  int diff; | for (int i=parent\_vec.size() -1 ; i>=0; i--)  {  node = parent\_vec.get(i);  diff = balanced (node);  if (diff == -2)  {  if (i>0)  rotate\_left(node,parent\_vec.get(i-1));  else  rotate\_left(node,NULL);  }  else if (diff == 2)  {  if (i>0)  rotate\_right(node,parent\_vec.get(i-1));  else  rotate\_right(node,NULL);  }}}  void AVLTree :: rotate\_left(AVLTreeNode \*parent, AVLTreeNode \*prev)  {  AVLTreeNode \*child;  child = parent -> getRight();  int diff = balanced (child);  if (diff == 1)  {  parent->setRight(child->getLeft());  child->setLeft(child->getLeft()->getRight());  parent->getRight()->setRight(child);  child = parent->getRight();  }  parent->setRight(child->getLeft());  child->setLeft(parent);  if (parent == root)  root = child;  else  {  if (prev->getKey() < child->getKey())  prev->setRight(child);  else  prev->setLeft(child);  }}  void AVLTree :: rotate\_right(AVLTreeNode \* parent, AVLTreeNode \* prev)  {  AVLTreeNode \*child;  child = parent -> getLeft();  int diff = balanced (child);  if (diff == -1)  {  parent->setLeft(child->getRight());  child->setRight(child->getRight()->getLeft());  parent->getLeft()->setLeft(child);  child = parent->getLeft();  }  parent->setLeft(child->getRight());  child->setRight(parent);  if (parent == root)  root = child;  else  {if (prev->getKey() < child->getKey())  prev->setRight(child);  else  prev->setLeft(child);  }}  ------------------------------------------------------------------  void AVLTree :: remove(int key)  {  AVLTreeNode \*tr;  vector <AVLTreeNode \*> parent\_vec;  tr = root;  while (tr->getKey() != key)  {  if (key > tr->getKey())  {  parent\_vec.push\_back(tr);  tr = tr->getRight();  }  else if (key < tr->getKey())  {  parent\_vec.push\_back(tr);  tr = tr->getLeft();  }  }  AVLTreeNode \*ptr;  ptr = tr;  AVLTreeNode \*pr;  if (tr == root)  {  if (tr -> getLeft() != NULL)  {  parent\_vec.push\_back(tr);  tr = tr->getLeft();  while (tr->getRight() != NULL)  {  parent\_vec.push\_back(tr);  tr = tr->getRight();  }  ptr->setKey(tr->getKey());  pr = parent\_vec.get(parent\_vec.size() -1);  if (pr->getKey() < tr->getKey())  pr->setRight(tr->getLeft());  else  pr->setLeft(tr->getLeft());  }  else  {  tr = tr->getRight();  root->setRight(NULL);  root = tr;  }  }  else  {  if (tr -> getLeft() != NULL)  {  parent\_vec.push\_back(tr);  tr = tr->getLeft();  while (tr->getRight() != NULL)  {  parent\_vec.push\_back(tr);  tr = tr->getRight();  }  ptr->setKey(tr->getKey());  pr = parent\_vec.get(parent\_vec.size() -1);  if (pr->getKey() < tr->getKey())  pr->setRight(tr->getLeft());  else  pr->setLeft(tr->getLeft());  }  else  {  pr = parent\_vec.get(parent\_vec.size() -1);  if (pr->getKey() < tr->getKey())  pr->setRight(tr->getRight());  else  pr->setLeft(tr->getRight());  }  }  AVLTreeNode \*node;  int left\_height;  int right\_height;  int diff;  for (int i=parent\_vec.size() -1 ; i>=0; i--)  {  node = parent\_vec.get(i);  diff = balanced (node);  if (diff == -2)  {  if (i>0)  rotate\_left(node,parent\_vec.get(i-1));  else  rotate\_left(node,NULL);  }  else if (diff == 2)  {  if (i>0)  rotate\_right(node,parent\_vec.get(i-1));  else  rotate\_right(node,NULL);  }  }}  **BTree Traversal**  void inordertraversal(BTreeNode \* curr, std::vector<int>& visit\_order)  {  std::vector<BTreeNode \*> stack;  std::vector<int> dir;  std :: vector<int> visited;  stack.push\_back(curr);  dir.push\_back(0);  if(curr->isLeaf()){  for(int i = 0; i < curr->getKeySize(); ++i){  visit\_order.push\_back(curr->getKey(i));  }  return;  }    while(stack.empty() == false)  {  int curr\_index = dir[dir.size()-1];  BTreeNode \* node = stack[stack.size()-1];  if(curr\_index < node->getChildSize())  {  BTreeNode \* child = (BTreeNode \*) node->getChild(curr\_index);  if(child->isLeaf())  {  for(int i = 0; i < child->getKeySize(); ++i)  {  visit\_order.push\_back(child->getKey(i));  }  if(curr\_index < node->getKeySize())  {  visit\_order.push\_back(node->getKey(curr\_index));  }  (dir[dir.size()-1])++;  }  else  {  if (dir[dir.size()-1] >0)  visit\_order.push\_back(node->getKey(curr\_index - 1));  (dir[dir.size()-1])++;  dir.push\_back(0);  stack.push\_back(child);  }}  else  {  if(curr\_index < node->getKeySize())  {  visit\_order.push\_back(node->getKey(curr\_index));  }  (dir[dir.size()-1])++;  dir.pop\_back();  stack.pop\_back();  }}}  -----------------------------------------------------------------------  **Btree Search**  BTreeNode \* BTree :: search(int key, std :: vector <BTreeNode \*> &parent\_vec,std :: vector<int> &pos\_vec, int \*index)  {  BTreeNode \*temp\_ptr = root;  while (temp\_ptr != NULL)  {  for (int i=0; i<temp\_ptr->keysize; i++)  {  if (key == temp\_ptr->key[i])  {  \*index = i;  return(temp\_ptr);  }  if (i < temp\_ptr->keysize-1)  {  if (temp\_ptr->key[i] < key && temp\_ptr->key[i+1] > key)  {  parent\_vec.push\_back(temp\_ptr);  pos\_vec.push\_back(i+1);  temp\_ptr = temp\_ptr->ptr[i+1];  break;  }  else if (temp\_ptr->key[i] > key)  {  parent\_vec.push\_back(temp\_ptr);  pos\_vec.push\_back(i);  temp\_ptr = temp\_ptr ->ptr[i];  break;  }  }  else  {  if (temp\_ptr->key[i] < key)  {  parent\_vec.push\_back(temp\_ptr);  pos\_vec.push\_back(i+1);  temp\_ptr = temp\_ptr->ptr[i+1];  break;  }  else if (temp\_ptr->key[i] > key)  {  parent\_vec.push\_back(temp\_ptr);  pos\_vec.push\_back(i);  temp\_ptr = temp\_ptr ->ptr[i];  break;  }}}}}  --------------------------------------------------------------------  If you are doing a bulk insert, you can use universal hashing to find a perfect hash function  h(x) = (((a \* num) + b) % p) % m  p = a prime greater than the range of num  m = a prime that is the table size  a = a randomly chosen constant greater than 1  b = a randomly chosen constant  ---------------------------------------------------------------  **Binary Tree from vec**  Create Perfect(sorted vec)  {  Queue low;  Queue high;  Low.push(0);  High.push (sorted vec.size -1);  Node \*root = NULL;  while (low.empty () == false)  {  int curr\_low = low.front();  int curr\_high = high.front();  int mid = (curr\_low + curr\_high)/2;  root = insertWithoutBall(root, sorted\_vec[mid]);  if (curr\_low < curr\_high){  low.push (curr\_low);  high.push(mid -1);  low.push(mid + 1);  high.push(curr\_high);  }}  **--------------------------------------------------------------------**  **-2, 1**  void rotate(grandparent, parent, curr)  {  AVLNode \*right = cuur->right;  curr->right = right->left;  right->left = curr;  right->right = parent;  if (gandparent->value < parent->value)  grandparent->right = right;  else  gradparent->left = right;  **--------------------------------------------------------------------**  Node \*cretaeperfecttree(head, len)  {  Node \* curr = head;  for (int i=0; i<len/2;++i)  { Node \*right = curr->right;  right->left = curr;  if (curr= head)  head= right;  curr= right->right;  }len= len/2;while(len > 1){  len =len/2;curr = head;  for(int i=0; i<len; ++i)  {  Node \*right = curr->right;  curr->right = right->left;  right->left = curr;if (curr == head)  head =right;curr= right->right;}  } |
|  |  |  |  |  |