# Algorithms Dictionary

Charlie Cook -- Algorithms & Data Structures -- 5/7/17

Last Updated 5/9/17 -- 30 Pages -- 177 KB

# Trees & Sorting

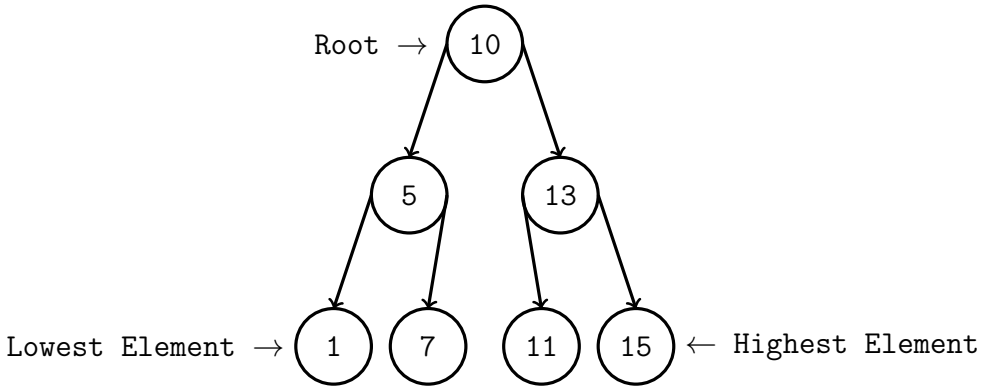
## Binary Search Tree

A B.S.T. is a data structure in which each element has one parent and (at most) two children; The lefthand children are always less than their parents and the righthand children are always greater than their parents. Only the location of the root/topmost element is needed to access the entire tree.

Starting from the root, go down all the lefthand children until there are no more. The lowest lefthand child is the smallest element in the tree. From there, go back to the lowest elements parent, then to the righthand child of said parent if it exists. Repeat until you arrive back at the root. From there, go to the first righthand child and go down its lefthand children until there are no more. Mirror the process described for the lefthand side of the tree, working your way down instead of up this time.

More simply, this can be visualized as walking around the tree in a counterclockwise rotation, starting at the root. Consider elements visited only when you are below them. In the below example, when you start at 10, you are above it. You cant be below 10 (map-wise) until you go beneath 1, 5, and 7.

Ex.




## Code

```
struct Node {
  int val;
  struct Node *left;
  struct Node *right;
};
```

```
typedef struct Node *Leaf;

//Creates a new node in memory, scans in its value, and sets its
   children to NULL
Leaf newLeaf() {
  Leaf temp;

  temp = (Leaf) malloc(sizeof(struct Node));

  scanf("%d", &(temp -> val));
  temp -> left = NULL;
  temp -> right = NULL;

  return temp;
}

//Calls newLeaf(), then determines the placement of the new node/
   leaf relative to the root node
void plantLeaf(Leaf root) {
  Leaf temp;
  int stopL, stopR;

  temp = newLeaf();

  stopL = 0;
  stopR = 0;

  while (!stopL && !stopR) {
    if (temp -> val < root -> val) {
      if (root -> left != NULL) {
        root = root -> left;
      } else {
        stopL++;
      }

    } else {
      if (root -> right != NULL) {
        root = root -> right;
      } else {
        stopR++;
      }
    }
  }

  if (stopL) {
    root -> left = temp;
  }
```

```
  if (stopR) {
    root -> right = temp;
  }
}

//Recursively goes around the tree in ascending order
void walk(Leaf leaf) {
  if (leaf != NULL) {
    walk(leaf -> left);

    printf("%d\t", leaf -> val);

    walk(leaf -> right);
  }
}

//Goes down the right children until none more exist to get the max
   value
void findMax(Leaf leaf) {
  if (leaf -> right != NULL) {
    findMax(leaf -> right);
  } else {
    printf("\nThe maximum leaf of this tree is %d.\n", leaf -> val);
  }
}

//Goes down the left children until none more exist to get the min
   value
void findMin(Leaf leaf) {
  if (leaf -> left != NULL) {
    findMin(leaf -> left);
  } else {
    printf("\nThe minimum leaf of this tree is %d.\n", leaf -> val);
  }
}

//Allows the user to explore the tree
void explore(Leaf leaf, Leaf root, int *choice) {
  if (leaf != NULL) {
    printf("\nYou are at %d.", leaf -> val);

    if (leaf == root) {
      printf(" (This is the root.)");
    }

    printf("\n[1]:\tGo left\n");
```

```
  printf("[2]:\tPeek left\n\n");
  printf("[3]:\tGo right\n");
  printf("[4]:\tPeek right\n\n");
  printf("[5]:\tStop exploring\n");

  printf("\nWhat do you want to do?\n\t--{ ");

  scanf("%d", choice);

  switch (*choice) {
    case 1: explore(leaf -> left, root, choice); break;
    case 2: {
      if (leaf -> left != NULL) {
        printf("\nTo your left, you can see a %d.\n", leaf -> left
            -> val);
      } else {
        printf("\nYou see nothing.\n");
      }
      explore(leaf, root, choice);
      break;
    }

    case 3: explore(leaf -> right, root, choice); break;
    case 4: {
      if (leaf -> right != NULL) {
        printf("\nTo your right, you can see a %d.\n", leaf ->
           right -> val);
      } else {
        printf("\nYou see nothing.\n");
      }
      explore(leaf, root, choice);
      break;
    }

    case 5: printf("Thanks for exploring this tree!\n"); break;
    default: printf("Hey! That's not a valid choice!\n");
  }
} else {
  printf("Dead end! Better luck next time!\n");
  printf("Wanna try again? [1 = Yes, 0 = No]: ");

  scanf("%d", choice);

  if (*choice) {
    explore(root, root, choice);
  } else {
    printf("Oh well, see ya later!\n");
```

```
    }
  }
}

//Recursivly searchs the tree for the target value
void search(Leaf leaf, int target) {
  if (leaf != NULL) {
    if (target < leaf -> val) {
      search(leaf -> left, target);
    } else if (target == leaf -> val) {
      printf("\nFound it! %d is in the tree!.\n", target);
    } else {
      search(leaf -> right, target);
    }
  } else {
    printf("\nSorry, %d is not in the tree.\n", target);
  }
}
```

Note: For the following five algorithms, this is the definition of the swap function:

```
void swap(int *g, int *h) {
  int temp = *g;
  *g = *h;
  *h = temp;
}
```

## Bubble Sort

Bubble Sort passes through an array as many times as there are elements in the array. In each pass, it checks every element against the one after it; If the after element is less than the current element, the elements are swapped. Once the last pass is complete, the array is sorted.

Bubble Sort can be optimized to have less passes than the number of elements by adding a check; If no swaps have occurred in a pass, the array is sorted.

### Ex.

```
array (no passes):  4, 2, 1, 3      [SWAPS: 0]
array (1 pass):     2, 1, 3, 4      [SWAPS: 3]
array (2 passes):   1, 2, 3, 4      [SWAPS: 4]
(The 3rd pass does not have any swaps. The list is sorted.)

(3 passes and 4 swaps were needed.)
```

### Code

```
void bubbleSort(int *array, int arraySize) {
  int m, n, swaps;
  printf("\nNow bubble sorting your array!\n");

  for (m = 0; m < arraySize - 1; m++) {

    swaps = 0;
    for (n = 0; n < arraySize - 1; n++) {

      if (array[n] > array[n + 1]) {
        swap(&array[n], &array[n + 1]);
        swaps++;
      }
    }

    if (!swaps) {
      if (m == 0) printf("\nHey! Your array was already sorted!\n");
      break;
    }
  }

  printf("Finished! Your array was bubble sorted!\n");
}
```

## Insertion Sort

Insertion Sort passes through an array once, starting from the second element. Each element is inserted into its proper position. This can be done by swapping the current element and its prior element while the prior element is greater than the current element. Once the pass is done, the array is sorted.

Ex.

```
array (no inserts): 4, 2, 1, 3
array (1 insert):   2, 4, 1, 3
array (2 inserts):  1, 2, 4, 3
array (3 inserts):  1, 2, 3, 4      [SORTED]

(1 pass and 3 swaps were needed.)
```

Code

```
void insertSort(int *array, int arraySize) {
  int i, j;

  printf("\nNow insertion sorting your array!\n");

  for (j = 1; j < arraySize; j++) {

    i = j - 1;

    while (i >= 0 && array[i] > array[i + 1]) {
      swap(&array[i + 1], &array[i]);
      i--;
    }
  }

  printf("Finished! Your array was insertion sorted!\n");
}
```

## Selection Sort

Selection Sort passes through an array once, starting from the second element. The minimum element of all elements beyond (and including) the current element is selected and moved down the array until it is at the top of the sorted partition. The moving down is accomplished by swapping the minimum element with elements before it until it is in the sorted partition, or the element before it is smaller than the minimum. Once the pass is done, the array is sorted.

### Ex.

```
array (no selects): 4, 2, 1, 3
array (1 select):   1, 4, 2, 3
array (2 selects):  1, 2, 4, 3
array (3 selects):  1, 2, 3, 4      [SORTED]

(1 pass and 3 swaps were needed.)
```

### Code

```
int minIdx(int *array, int arraySize, int firstIndex) {
  int i, min;
  min = firstIndex;

  for (i = firstIndex; i < arraySize; i++) {

    if (array[i] < array[min]) {
      min = i;
    }
  }

  return min;
}

void selectSort(int *array, int arraySize) {
  int i, m;
  printf("\nNow selection sorting your array!\n");

  for (i = 1; i < arraySize; i++) {
    m = minIdx(array, arraySize, i);

    while (m >= i && array[m] < array[m - 1]) {
      swap(&array[m], &array[m - 1]);
      m--;
    }
  }
  printf("Finished! Your array was selection sorted!\n");
}
```

## Merge Sort

Merge Sort subdivides an array recursively until each partition is a single element. It then merges the one element arrays back together in order, yielding a sorted array at the end.

Ex.

```
array:  4, 2, 1, 3

part. 1:    4, 2
subparts:   [4], [2]
merged: 2, 4

part. 2:    1, 3
subparts:   [1], [3]
merged: 1, 3

merged: 1, 2, 3, 4
```

Code

```
void merge(int *targ, int *a, int *b, int sizeA, int sizeB) {
  int i, j, k;

  i = 0;
  j = 0;

  for (k = 0; k < sizeA + sizeB; k++) {

    if ((i < sizeA && a[i] < b[j]) || j == sizeB) {
      targ[k] = a[i];
      i++;
    } else {
      targ[k] = b[j];
      j++;
    }
  }
}

void mergeSort(int *array, int arraySize) {
  int *f, *g;
  int sizeF, sizeG, i, j;

  sizeF = arraySize / 2;
  sizeG = arraySize - sizeF;

  if (arraySize > 1) {

    j = 0;
```

```
    f = (int *) calloc(sizeF, sizeof(int));
    for (i = 0; i < sizeF; i++) {

      f[i] = array[j];
      j++;
    }

    g = (int *) calloc(sizeG, sizeof(int));
    for (i = 0; i < sizeG; i++) {

      g[i] = array[j];
      j++;
    }

    mergeSort(f, sizeF);
    mergeSort(g, sizeG);

    merge(array, f, g, sizeF, sizeG);
  }
}
```

## Heap Sort

Heap Sort takes the elements in an array and moves them into a heap, a tree structure where each element has at most two children. The root of the tree is the maximum element from the array, and each child is smaller than its parent. Elements in the heap can be swapped easily to keep the tree in heap form, and the maximum is always on top. To sort, simply take the root out and put it at the end of the sorted array, then reorder the heap. The heap will shrink until only one element remains.

### Ex.

```
array (pre-heap):   4, 2, 1, 3
heap (0 removals):    4      sorted array:
                     /\
                    3 1
                    /
                    2
heap (1 removal):     3      sorted array: 4
                     /\
                    2 1
heap (2 removals):    2      sorted array: 3, 4
                      \
                      1
heap (3 removals):  1        sorted array: 2, 3, 4
sorted array (post-heap):    1, 2, 3, 4

(1 heapify, 3 reorders, and 4 removals were needed.)
```

### Code

```
void heapify(int *array, int head, int heapSize) {
  int left, right, largest;

  left = 2 * head + 1;
  right = 2 * head + 2;
  largest = head;

  if (left < heapSize && array[left] > array[head]) {
    largest = left;
  }
  if (right < heapSize && array[right] > array[largest]) {
    largest = right;
  }

  if (largest != head) {
    swap(&array[largest], &array[head]);
  }
}
```

```
void buildHeap(int *array, int arraySize) {
  int i;

  for (i = arraySize / 2; i >= 0; i--) {

    heapify(array, i, arraySize);
  }
}

void heapSort(int *array, int arraySize) {
  int i;

  printf("\nNow heap sorting your array!\n");

  buildHeap(array, arraySize);
  printHeap(array, arraySize);

  for(i = arraySize - 1; i > 0; i--) {

    swap(&array[0], &array[i]);
    buildHeap(array, i);

    printHeap(array, i);
  }

  printf("Finished! Your array was heap sorted!\n");
}
```

# Time Complexity

Time Complexity is a way to classify and rank algorithms. As the name implies, the runtime of an algorithm is what is used in ranking. An algorithm's runtime can be represented as some function of the size of the input data, commonly designated as $n$. For sorting algorithms, $n$ is the number of elements in the array that needs sorting.

To simplify the expressions, Big-O Notation is used, which drops out the lower order terms and leading coefficient in favor of focusing on the leading term. Below is a table of time complexity classes and algorithms associated with those classes.

| Time Type | Big-O Type | Algorithm(s) |
|---|---|---|
| Constant | $O(1)$ | Swapping Two Array Elements |
| Logarithmic | $O(\log(n))$ | Binary Search |
| Linear | $O(n)$ | Printing an Array, Finding Array Extrema |
| Linearithmic | $O(n\log(n))$ | Merge & Heap Sort |
| Quadratic | $O(n^2)$ | Bubble, Insertion, & Selection Sort |
| Factorial | $O(n!)$ | Bogo Sort (See the next page) |

Time Complexity Table

# The Traveling Salesman Problem

The Traveling Salesman Problem is an open NP-Hard problem in Computer Science. It simply states:

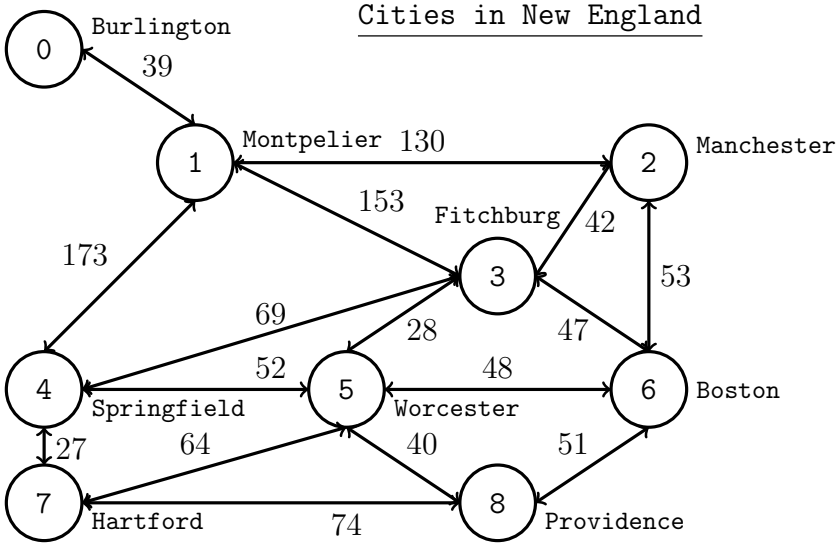
Given a list of cities and the distances between them, what is the shortest route that will let you visit all cities and return to the city you start from?

An alternate version of it, designated the T.S.P. Decision Version, also exists, which states:

Given a limit to the distance of a route, does a viable route exist that will satisfy T.S.P. and be under the limit?

If we reference the graph below, say we want to start from Fitchburg (node 3) and visit all of the eight cities in the New England area that are shown. If we consider the normal version of T.S.P., the first thing we might try is to list out all routes that visit all cities and end up back in Fitchburg. This might take a while, and if we also wanted to visit some small towns as well, say we add 20 of them to the graph, the number of routes will grow worse than exponentially.

In fact, attempting this method of brute-forcing T.S.P. has a time complexity of $O(n!)$, where $n$ is the number of cities. Bogo Sort also has this time complexity. For those unaware, Bogo Sort involves randomly shuffling a list, checking if its sorted, and repeating until the list is sorted. Clearly, brute-force will not be efficient for this problem, hence it's classification as NP-Hard.



# Dijkstra & Pathfinding

Dijkstra's Shortest Path Algorithm is fairly straightforward. First, you need a graph/map, which consists of a group of nodes linked in a certain fashion. The nodes should be numbered starting from 0, and can be represented as a square matrix or two-dimensional array. In this, a row corresponds to the paths that lead out of a node, and a column corresponds to the paths that lead into a node.

To find the shortest path through the graph or from a given node to every other, do the following:

1. Create three arrays as long as the number of nodes in the graph. Name them "distance", "accessed", and "previous"

2. Initialize the distance and previous arrays to be all -1, except the index corresponding the the origin node. For that node, set its distance to 0 and previous to its index in the matrix

3. Initialize the accessed array be all 0, except the origin node, which should be 1.

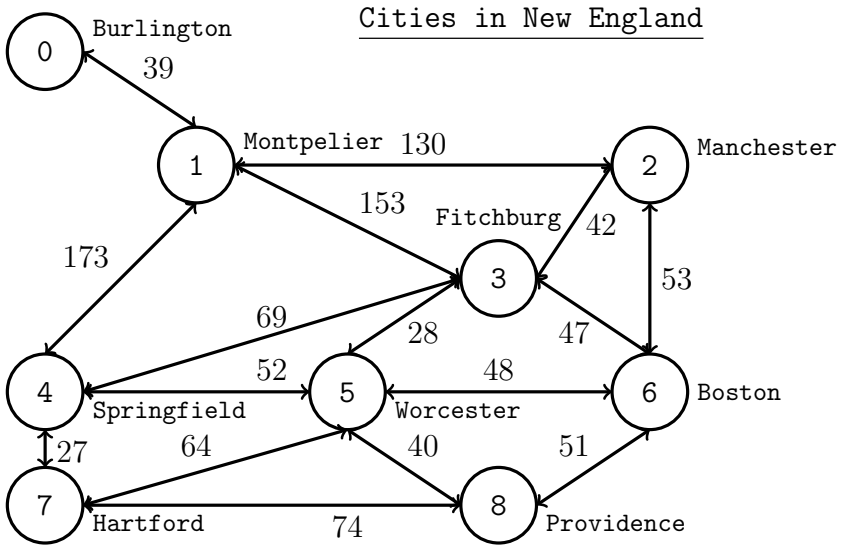
4. From the origin node, scan through its row in the matrix. If the distance to a new node plus the distance to the origin node is less than the current distance to the new node, or there is a path to the new node and no path to it currently exists (i.e. it has a distance value of -1), update the distance to the new node and its entry in the previous array.

5. Once the row of the origin node is scanned, select the node that has the smallest distance and is also unaccessed. Move to this node, and set it as accessed.

6. Repeat steps 4 and 5 until all nodes have been accessed. The final versions of the distance and previous arrays will allow you to extrapolate the shortest paths to any node.

To read a path to a node, go to its entry in the previous array. Read what node was prior to your target node, and check its prior node. Repeat this until you find a node who is its own prior node, which corresponds to the origin node. Then, start printing the indicies of the nodes from origin to target.

Ex.




(NE.txt)

```
9

0   39  -1  -1  -1  -1  -1  -1  -1
39  0   130 153 173 -1  -1  -1  -1
-1  130 0   42  -1  -1  53  -1  -1
-1  153 42  0   69  28  47  -1  -1
-1  173 -1  69  0   52  -1  27  -1
-1  -1  -1  28  52  0   48  64  40
-1  -1  53  47  -1  48  0   -1  51
-1  -1  -1  -1  27  64  -1  0   74
-1  -1  -1  -1  -1  40  51  74  0
```

(NE_Solved.txt)

```
INDEX:  0       1       2       3       4       5       6       7       8
        ------  ------  ------  ------  ------  ------  ------  ------  ------
DIST:   |192    |153    |42     |0      |69     |28     |47     |92     |68     |
ACCS:   |1      |1      |1      |1      |1      |1      |1      |1      |1      |
PREV:   |1      |3      |3      |3      |3      |3      |3      |5      |5      |

The shortest path to node 0 from node 3 is 1, 0.
The shortest path to node 1 from node 3 is 1.
The shortest path to node 2 from node 3 is 2.
The shortest path to node 4 from node 3 is 4.
The shortest path to node 5 from node 3 is 5.
The shortest path to node 6 from node 3 is 6.
The shortest path to node 7 from node 3 is 5, 7.
The shortest path to node 8 from node 3 is 5, 8.
```

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//Coded by Charlie C. | Started 4/16/17

int* getLine(int size, FILE *in);
int* prepareDP(int size, int x, int *a);
void printLines(int size);
void writeLines(int size, FILE *fo);
void showDAP(int size, int *d, int *a, int *p);
void writeDAP(int size, int *d, int *a, int *p, FILE *fo);
void showMap(int size, int **map);
int checkACCS(int size, int *a);
int* updateDist(int size, int node, int *d, int *p, int **m);
int getMinDist(int size, int* d, int *a);
int* getPath(int size, int *p, int *path, int node, int i);
void writePath(int size, int node, int origin, int *path, FILE *fo);

//Main function
int main(){
  //Declarations for some ints, the D.A.P. arrays, and the map
     matrix
  int mapSize, i, origin, node, *dist, *accs, *prev, *pathBuff, **
     map;
  //Char buffer for taking in the name of the input file
  char fBuffer[64];
  //File pointers for input and output
  FILE *fIn, *fOut;

  printf("Enter the name of the file containing your map: ");
  scanf("%s", fBuffer);

  fIn = fopen(fBuffer, "r");
  //Gets the number of nodes in the map,
  //which is the first entry in the input file
  fscanf(fIn, "%d", &mapSize);

  //Allocate memory for the dist array
  dist = (int*) calloc(mapSize, sizeof(int));

  //Allocate memory for the accs array
  accs = (int*) calloc(mapSize, sizeof(int));

  //Allocate memory for the prev array
  prev = (int*) calloc(mapSize, sizeof(int));
```

```
//Allocates space for the pointers to the arrays
//that make up the rows of the matrix
map = (int**) calloc(mapSize, sizeof(int));

//Iteratively read in the map data to fill the matrix
for (i = 0; i < mapSize; i++) {

  map[i] = getLine(mapSize, fIn);
}

fclose(fIn);

showMap(mapSize, map);

printf("Map Loaded!\n");

do {
  printf("Enter the node you want to start from:\n(Must be an
     integer from 0 to %d)\n(If your map is not doubly-linked,
     input the origin node)\n\t--{ ", mapSize - 1);
  scanf("%d", &node);

  if (node < 0 || node > mapSize - 1) {
    printf("\nError: Invalid Node.\n");
  }
} while (node < 0 || node > mapSize - 1);

//Set all values after the initial to -1
prepareDP(mapSize, node, dist);

accs[node] = 1;

//Set all values after the initial to -1
prepareDP(mapSize, node, prev);
prev[node] = node;

//Stores the origin node somewhere safe
origin = node;

//Show the initial D.A.P. arrays and start node
showDAP(mapSize, dist, accs, prev);
printf("\nNEXT NODE:\t%d\n", node);

//We should keep going to nodes that aren't accessed
//Thus, checking if accs is filled will be our condition
while(checkACCS(mapSize, accs)) {
  updateDist(mapSize, node, dist, prev, map);
```

```
    i = getMinDist(mapSize, dist, accs);
    accs[i] = 1;
    node = i;

    showDAP(mapSize, dist, accs, prev);
    printf("\nNEXT NODE:\t%d\n", node);

  }

  printf("\nDone!\nEnter the base name of the output file you want
     for the D.A.P. arrays: ");
  scanf("%s", fBuffer);
  strcat(fBuffer, "_Solved.txt");

  fOut = fopen(fBuffer, "w");
  writeDAP(mapSize, dist, accs, prev, fOut);

  fprintf(fOut, "\n");

  pathBuff = (int*) calloc(mapSize, sizeof(int));
  prepareDP(mapSize, mapSize, pathBuff);

  for (i = 0; i < mapSize; i++) {
    if (i != origin) {
      getPath(mapSize, prev, pathBuff, i, 1);
      writePath(mapSize, i, origin, pathBuff, fOut);
      prepareDP(mapSize, mapSize, pathBuff);
    }
  }

  fclose(fOut);

  printf("\nCheck the output file for the shortest paths.");

  //showDAP(mapSize, dist, accs, prev);

  return 0;
}

//Scans a line from the input file and returns it as a pointer array
int* getLine(int size, FILE *in) {
  int i;
  int *line;

  line = (int*) calloc(size, sizeof(int));

  for (i = 0; i < size; i++) {
```

```
    fscanf(in, "%d", &line[i]);
  }

  return line;
}

//Sets all cells after the initial in an array to -1
int* prepareDP(int size, int x, int *a) {
  int i;

  if (size > 1) {

    for (i = 0; i < size; i++) {

      if (i != x) {
        a[i] = -1;
      }
    }
  }

  return a;
}

//Prints out lines when showing the contents of the D.A.P. arrays
void printLines(int size) {
  int i;

  printf("\n\t");

  for (i = 0; i < size; i++) {
    printf("------\t");
  }
}

//Writes lines when writing the contents of the D.A.P. arrays to the
    output file
void writeLines(int size, FILE *fo) {
  int i;

  fprintf(fo, "\n\t");

  for (i = 0; i < size; i++) {
    fprintf(fo, "------\t");
  }
}

//Shows the contents of the D.A.P. arrays
```

```
void showDAP(int size, int *d, int *a, int *p) {
  int i;

  printf("\nINDEX:\t");

  for (i = 0; i < size; i++) {

    printf("%d\t", i);
  }

  printLines(size);
  printf("\nDIST:\t");

  for (i = 0; i < size; i++) {

    printf("|%d\t", d[i]);
  }

  printf("|\nACCS:\t");

  for (i = 0; i < size; i++) {

    printf("|%d\t", a[i]);
  }

  printf("|\nPREV:\t");

  for (i = 0; i < size; i++) {

    printf("|%d\t", p[i]);
  }

  printf("|\n");
}

//Writes the contents of the D.A.P. arrays to the output file
void writeDAP(int size, int *d, int *a, int *p, FILE *fo) {
  int i;

  fprintf(fo, "\nINDEX:\t");

  for (i = 0; i < size; i++) {

    fprintf(fo, "%d\t", i);
  }

  writeLines(size, fo);
```

```
  fprintf(fo, "\nDIST:\t");

  for (i = 0; i < size; i++) {

    fprintf(fo, "|%d\t", d[i]);
  }

  fprintf(fo, "|\nACCS:\t");

  for (i = 0; i < size; i++) {

    fprintf(fo, "|%d\t", a[i]);
  }

  fprintf(fo, "|\nPREV:\t");

  for (i = 0; i < size; i++) {

    fprintf(fo, "|%d\t", p[i]);
  }

  fprintf(fo, "|\n");
}

//Shows the contents of the map matrix
void showMap(int size, int **map) {
  int i, j;

  printf("\nMAP:\n");

  for (i = 0; i < size; i++) {

    for (j = 0; j < size; j++) {

      printf("|%d\t", map[i][j]);
    }

    printf("|\n");
  }
}

//Returns a boolean value based on the accs array's state
int checkACCS(int size, int *a) {
  int i;

  for (i = 0; i < size; i++) {
```

```
    if (a[i] == 0) {
      return 1;
    }
  }

  return 0;
}

//Updates the dist and prev arrays relative to a single node
int* updateDist(int size, int node, int *d, int *p, int **m) {
  int i;

  for (i = 0; i < size; i++) {

    if (m[node][i] != -1 && (m[node][i] + d[node] < d[i] || d[i] ==
       -1)) {
      d[i] = m[node][i] + d[node];
      p[i] = node;
    }
  }
}

//Finds the index containing the minimum unaccessed distance
int getMinDist(int size, int* d, int *a) {
  int i, min;

  min = -1;

  for (i = 0; i < size; i++) {
    //If the ith element hasn't been accessed and
    //If the distance for the ith element has been set and
    //If the ith distance is less than current min distance or
    //If the min is uninitialized
    if ((a[i] == 0 && d[i] > -1) && (d[i] < d[min] || min == -1)) {
      min = i;
    }
  }

  return min;
}

//Gets the shortest path to a node once travelling is complete.
int* getPath(int size, int *p, int *path, int node, int i) {
  if (p[node] != node) {
    getPath(size, p, path, p[node], i + 1);
    path[size - i] = node;
  }
```

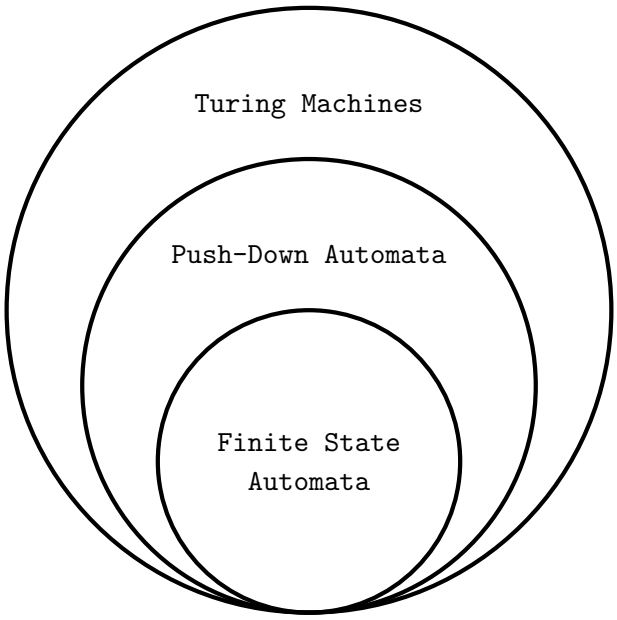
```
  return path;
}

//Writes out the shortest path to the output file
void writePath(int size, int node, int origin, int *path, FILE *fo)
   {
  int i;

  fprintf(fo, "The shortest path to node %d from node %d is ", node,
      origin);

  for (i = 0; i < size; i++) {
    if (path[i] != -1) {
      fprintf(fo, "%d", path[i]);
      if (i + 1 == size) {
        fprintf(fo, ".\n");
      } else {
        fprintf(fo, ", ");
      }
    }
  }
}
```

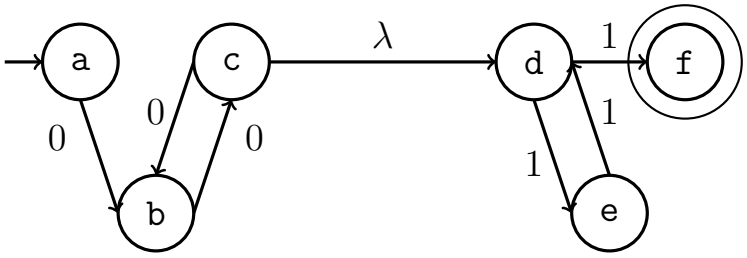
# Theory of Computation




Simplified Chomsky Hierarchy

## Finite State Automata

Finite State Automata, found in the center of Chomsky's Hierarchy (pictured above), are the simplest forms of computers. They can be visualized as a graph of nodes, where the start node has a blank arrow leading into it and where the end node is a double-circle. For example, below is a F.S.T. that can generate all strings that have an even number of 0's followed by an odd number of 1's. Please note that arrows marked with a $\lambda$ are null paths, which do not produce anything.

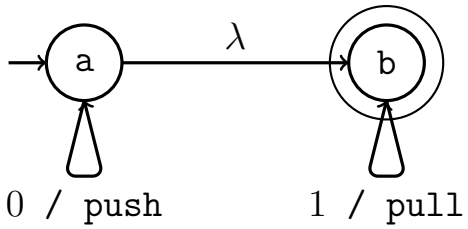



F.S.A.: $0^n1^m : n$ is even, $m$ is odd

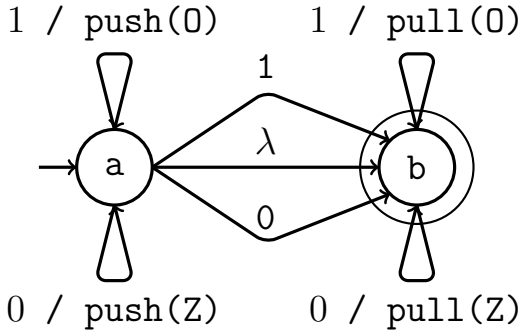
## Push-Down Automata

Push-Down Automata, which encompass Finite State Automata, implement a stack as a crude form of memory. The concept of a stack is critical to Computer Science, as it can represent recursion and other tricky concepts in an elementary way. A stack can have objects pushed onto it or popped off of it. Only the top element on a stack, the element last pushed on, can be popped off. Though physical implementations of stacks generally have a limit to their size, for our purposes, assume stacks can fit near-infinitely many elements.

One thing that Push-Down Automata can do that Finite State Automata cannot is generate all strings that have an equal number of 0's and 1's. Also, Push-Down Automata can generate all binary palindrome strings. Examples of both of these can be seen below. It should be noted that in order for an automata to finish, its stack must be empty. Also, if a push invovles a letter, that letter must be at the top of the stack in order for a pull of it to be processed. Finally, as with Finite State Automata, $\lambda$ paths do not do anything.




P.D.A.: $0^n1^n : n \in \mathbb{N}$




P.D.A.: Binary Palindromes

## Turing Machines

Turing Machines, which encompass the prior two automata types, can compute the answer to any problem for which an algorithm exists. Pioneered and commemorated to Alan Turing, the Father of Computer Science, they function as such:

Using a (reasonably) infitine tape of cells, a Turing Machine is instructed with a set of rules. For each rule, the T.M. reads the current cell it is at on the tape. Based on what is read, the T.M. may change to a new state, and may overwrite the cell. Once a state-change and/or a write occur, the T.M. will then move one cell to the left or to the right. Eventually, a properly designed Turing Machine will enter a halt state, in which it can be in an "accept" substate, or in a "reject" substate.

The concept of the state, coupled with the tape of data and instructions, paved the way for the development of general purpose computers following the Second World War in the mid 20th Century. Unfortunately, due to homophobic sentiments and nosy police officers, Alan Turing was forced into hormone theropy when it was revealed he was gay, a crime under contemporary British law. This, coupled with the resentment Turing now faced in the industry, led him to end his life prematurely, and he would not be pardoned by the government of Britain for nearly five decades afterwards. Back on topic.

Below are instructions for a Turing Machine that will increment a binary number on a tape. Each instruction should be read as:

[Current State], [Data Read]=>[New State], [Data Write], [Move Tape]

S is the starting state. $\delta$ is the leftmost cell on the tape, and beyond which the machine is barred from going. _ is a blank cell. Once the "accept" state is entered, the machine halts.

S, $\delta$=>S, $\delta$, R;
S, 0=>Z, 0, R;
S, 1=>$\Omega$, 1, R;

$\Omega$, 1=>$\Omega$, 1, R;
$\Omega$, 0=>Z, 0, R;
$\Omega$, _=>F, 0, L;

Z, 0=>Z, 0, R;
Z, 1=>Z, 1, R;
Z, _=>A, _, L;

A, 0=>H, 1, L;
A, 1=>A, 0, L;

H, 0=>H, 0, L;
H, 1=>H, 1, L;
H, $\delta$=>accept;

F, 1=>F, 0, L;
F, $\delta$=>W, $\delta$, R;

W, 0=>H, 1, L;

### T.M.: Binary Incrementer with Overflow Accommodation

# RSA Encryption

Note: the traditional Alice and Bob characters will be supplemented by Amuro Ray and Sayla Mass from Mobile Suit Gundam.

Amuro and Sayla wish to talk securely, since they know Char, Sayla's over-protective and slightly fascist older brother, is listening to them. First, they each pick two prime numbers (usually large but for this example they will be small) to keep secret.

Amuro's primes, $P_A$ and $Q_A$, are 5 and 29.
Sayla's primes, $P_S$ and $Q_S$, are 7 and 23.

Amuro transmits the product of his primes, $N_A = 145 (= 5 * 29)$.
Sayla transmits the product of her primes, $N_S = 161 (= 7 * 23)$.

Char can see both $N_A$ and $N_S$. So too can Sayla and Amuro.

Amuro picks another prime number, $E_A = 13$, and transmits it.
Sayla picks another prime number, $E_S = 17$, and transmits it.

Char can see both $E_A$ and $E_S$. So too can Sayla and Amuro. $E_A$ and $E_S$ act as Amuro and Sayla's public/encryption keys. Now Amuro and Sayla can send each other encrypted messages.

Using ASCII, Amuro encrypts his message, $M_A$ = "X", or 88 in ASCII, using the following formula:

$$M_A{}^{E_S} \mod N_S = 88^{17} \mod 161 = 44 = \text{","} = C_A$$

$C_A$ is Amuro's ciphertext, which he can transmit. He does.

Sayla chooses her message, $M_S$ = "O", or 79 in ASCII, and uses a similar formula:

$$M_S{}^{E_A} \mod N_A = 79^{13} \mod 145 = 69 = \text{"E"} = C_S$$

$C_S$ is Sayla's ciphertext, which she can transmit. She does.

Char can see Amuro and Sayla's ciphertexts, "," and "E", and is perplexed.

To decrypt $C_S$, Amuro must calculate his private/decryption key, $D_A$. He can do this because of the following congruence:

$$E_A * D_A \equiv 1 \mod (P_A - 1) * (Q_A - 1)$$

This translates to the equation:

$$E_A * D_A = k * (P_A - 1) * (Q_A - 1) + 1$$

Where $k$ is some integer. Solving this equation for $D_A$ shows:

$$\begin{aligned} D_A &= (k * (P_A - 1) * (Q_A - 1) + 1)/E_A \\ &= (k * (5 - 1) * (29 - 1) + 1)/13 \\ &= (k * 4 * 28 + 1)/13 \\ &= (112k + 1)/13 \end{aligned}$$

$D_A$ needs to be an integer, and $k = 8$ results in $D_A = 69$.

Amuro can put $C_S$ through a similar equation now to get $M_S$:

$$C_S{}^{D_A} \mod N_A = 69^{69} \mod 145 = 79 = M_S = O$$

Likewise, Sayla can compute her private key, $D_S$:

$$\begin{aligned} D_S &= (k * (P_S - 1) * (Q_S - 1) + 1)/E_S \\ &= (k * (7 - 1) * (23 - 1) + 1)/17 \\ &= (k * 6 * 22 + 1)/17 \\ &= (132k + 1)/17 \end{aligned}$$

$D_S$ needs to be an integer, and $k = 13$ results in $D_S = 101$.

Now Sayla can decrypt $C_A$ to get $M_A$:

$$C_A{}^{D_S} \mod N_S = 44^{101} \mod 161 = 88 = M_A = X$$

All the while, Char is none the wiser to Amuro and Sayla's messages, since Char doesn't know $P_A$, $Q_A$, $P_S$, or $Q_S$, and this cannot calculate $D_A$ or $D_S$.

If Amuro wishes to send his signature as well, and keep it secret, he first encrypts his signature, $S_A$ = "A", or 65 in ASCII, with his decryption/private key, $D_A$. Next, he encrypts the resulting message with Sayla's encryption/public key, $E_S$. The resulting ciphered-signature will be referred to as $CS_A$.

$$\begin{aligned}(S_A{}^{D_A} \mod N_A)^{E_S} \mod N_S = \\ (65^{69} \mod 145)^{17} \mod 161 = 156 = CS_A = \text{"£"}\end{aligned}$$

(Note: The symbol for $CS_A$ is from extended ASCII)

Amuro can now transmit $CS_A$. When Sayla receives it, she knows to decrypt it first with her decryption/private key, $D_S$, then with Amuro's encryption/public key, $E_A$.

$$\begin{aligned}(CS_A{}^{D_S} \mod N_S)^{E_A} \mod N_A = \\ (156^{101} \mod 161)^{13} \mod 145 = 65 = S_A = \text{"}A\text{"}\end{aligned}$$

Since Sayla received Amuro's initial "A", his signature, from decrypting the message with Amuro's encryption/public key, $E_A$, she can trust that it must have been encrypted with Amuro's decryption/private key, $D_A$, which only he has access to.