|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Intro Algorithms** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| *Finding a value in an array* | | | | | | | | | | | | | | | *Box of 0s and 1s* | | | | | | | | | | | | | | |
| Find searchval in array [O(*n*)] | | | Binary search [O(lg*n*)] | | | | | | | | | | | | Individually [O(*n*2)] | | | | Secondary algorithm [O(*n*)] | | | | | | | | | | |
| int search(int\* array, int length, int searchval) {  int i;  for (i=0; i<length; i++)  if (array[i] == searchval)  return 1;  return 0;  } | | | int binsearch(int\* array, int length, int searchval) {  int low = 0, high = length-1;  while (low <= high) {  int mid = (low+high)/2;  if (searchval < array[mid]) high = mid-1;  else if (searchval > array[mid]) low = mid+1;  else return 1;  }  return 0;  } | | | | | | | | | | | | Check the number of 1s row by row.  Return the one that has the most. | | | | int maxOnes(int\*\* grid, int n, int m) {  int i = 0, j = 0;  while (i < n && j < m) {// Go until we're out of bounds.  while (j < m && grid[i][j] == 1) j++; // Walk down row until a 0  i++; // Go to the next row  }  return j;  } | | | | | | | | | | |
| **Sorted List Matching Problem** | | | | | | | | | | | | | | | | | | | | | | | | | | **Files** | | | |
| Brute Force [O(*n2*)] | | | | | Binary search [O(*n*lg*n*)] | | | | | | | | | | | Alternative algorithm [O(*n*)] | | | | | | | | | | Reading  FILE \* fin = fopen("file.txt", "r");  if (fin != NULL && feof(fin) == 0)  fscanf(fin, "%d", &size);  Writing  FILE \* fout = fopen("file.txt", "w");  if (fp != NULL)  fprintf(fout, "%d", size);  Closing  fclose(fin);  fclose(fout); | | | |
| void printMatches(char list1[][SIZE],  char list2[][SIZE],  int len1, int len2) {  int i,j;  for (i=0; i<len1; i++) {  for (j=0; j<len2; j++) {  if (strcmp(list1[i],list2[j]) == 0) {  printf(“%s\n”, list1[i]);  break;  }  }  }  } | | | | | int binSearch(char list[][SIZE], int len, char name[]) {  int low = 0, high = len-1;  while (low <= high) {  int mid = (low+high)/2, cmp = strcmp(name, list[mid]);  if (cmp < 0) high = mid-1;  else if (cmp > 0) low = mid+1;  else return 1;  }  return 0;  }  int rbinsearch(int low, int high, int searchVal, int array[]) {  if (low > high) return 0;  int mid = (low+high)/2;  if (searchVal < array[mid])  return binsearch(low, mid-1, searchVal, array);  else if (searchVal > array[mid])  return binsearch(mid+1, high, searchVal, array);  else return 1;  } | | | | | | | | | | | 1) Start 2 markers at beginning of both lists.  2) Repeat until a marker reaches end of its list:  a) Compare the two names that the markers are pointing at.  b) If they are equal, output the name and advance BOTH markers one spot.  If they not equal, advance the marker pointing to the name that comes earlier alphabetically one spot. | | | | | | | | | |
| for (i=0; i<size1; i++) {  int found = 0;  for (j = 0; j < size2; j++) {  if (list1[i] == list2[j]) {  found = 1;  break;  }  }  if (found) printf("%d ", list1[i]); | | | | | | | | | |
| **Recurrence Relations** | | | |
| T(n) = 2T(n-1) + 1  = 2[2T(n-2) + 1] + 1 = 4T(n-2) + 3  = 4[2T(n-3) + 1] + 3 = 8T(n-3) + 7 | | | |
| **Dynamic Memory Allocation BG** | | | |
| Enumeration Types | | | |
| typedef enum {penny = 1, nickel = 5, dime = 10, quarter = 25, halfdollar = 50} coinT; | | | |
| **Dynamic Memory Allocation** | | | | | | | | | | | | | | | | | | | | | | | | | | **Data and Memory** | | | |
| malloc() | ([type])malloc([n] \* sizeof([type]); | | | | | | | | | | | | Returns NULL when fail | | | | | | | | | | | | | Bit = 0 or 1 | | | |
| calloc() | ([type])calloc([n], sizeof([type]); | | | | | | | | | | | | Initializes values to 0 | | | | | | | | | | | | | Byte = 8 bits [char] | | | |
| realloc() | ([type])realloc(variable, (size+EXTRA) \* sizeof([type])); | | | | | | | | | | | | Reallocates EXTRA memory | | | | | | | | | | | | | Word = 2-4 bytes [int] | | | |
| free() | free(variable); | | | | | | | | | | | | For every malloc() there is an equal and opposite free() | | | | | | | | | | | | |  | | | |  |
| **Base Conversion** | | | | | | | | | | | | | | | | | | | | | | | DX and DY arrays | | | | | | |
| Base b 🡪 Decimal | | | | | | | | | | | Decimal 🡪 Base b | | | | | | | | | | | | const int DX[] = {-1,-1,-1,0,0,1,1,1};  const int DY[] = {-1,0,1,-1,1,-1,0,1};  int temp\_x += DX[i];  int temp\_y += DY[i]; | | | | | | |
| dn-1dn-2…d2d1d0 (b) = dn-1xbn-1 + dn-2xbn-2 + … + d2xb2 + d1xb + d0 | | | | | | | | | | | Divide decimal number by *b* continually.Read remainders backwards. | | | | | | | | | | | |
| 3145 = 3 x 52 + 1 x 51 + 4 x 50 = 8410.  A3D16 = Ax162 + 3x161 + D = 10x162 + 3x16 + 13 = 262110. | | | | | | | | | | | [x % *b*][x / *b*) % *b*] backwards | | | | | | | | | | | |
| **Summation Rules & Arithmetic/Geometric Series** | | | | | | | | | | | | | | | | | | | | | | | | | **Algorithm Analysis** | | | | |
|  | |  | | | | | | | |  | | | | | | | |  | | | | | | | * + T(*n*) = cf(*n*)   + [Reasonable: O(*nk*), O(*n*lg*n*)] [Unreasonable: O(c*n*)]   + for(i = 0; i < n; i++) & while(i < n) loops take *n*     - Nested loops take *nm* if inner runs every time.     - Check which loops run   + while(*n* < k) loops that update *n* 🡪 *n*/*b* take   k = log*bn* steps (i.e., O(lg*n*)) | | | | |
|  | | , | | | | | | | |  | | | | | | | |  | | | | | | |
| **Recursion Intro, Towers of Hanoi, Fast Exponentiation, & Floodfill** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **if (terminating condition) DO FINAL ACTION**  **else {**  **Take one step closer to terminating condition**  **Call function RECURSIVELY on smaller subproblem**  **}** | | | | | | | | | | | | **if (!(terminating condition) ) {**  **Take a step closer to terminating condition**  **Call function RECURSIVELY on smaller subproblem**  **}** | | | | | | | | | | | | int DigitalRoot(int n) {  int sumd = DigitSum(n);  if (sumd > 9)  return DigitalRoot(sumd);  return sumd;  } | | | | | |
| int Power(int base, int exp) { //O(lg exp)  if ( exp == 0) return 1;  else  return (base\*Power(base, exp-1));  } | | | | int bin\_coeff(int n, int k) {  if ((k == 0) || (n == k) return 1;  else  return bin\_coeff(n-1, k-1) + bin\_coeff(n-1, k);  } | | | | | | | | #define TIP\_RATE  void Tip\_Chart (int first\_val, int last\_val)  if (!(first\_val > last\_val) ) {  printf(“On a meal of $%d”, first\_val);  printf(“ you should tip $%lf\n”, first\_val\*TIP\_RATE);  Tip\_Chart(first\_val + 1, last\_val);  }  } | | | | | | | | | | | | int DigitSum(int n) {  if (n > 0) return n%10 + DigitSum(n/10);  return 0;  } | | | | | |
| int Fib(int n) {  if (n < 3) return 1;  else return (Fib(n-1) + Fib(n-2));  } | | | | int Lucas(int n) {  if (n == 1) return 1;  else if (n == 2) return 3;  else return Lucas(n-1)+Lucas(n-2);  } | |
| int Triangle\_Number(int n) {  if ( n == 0 ) return 0;  else return (n + Triangle\_Number(n-1));  } | | | | | | int fact(int n) {  if (n == 0) return 1;  else return (n \* fact(n-1));  } | | | | | |
| int binSearch(int \*values, int low, int high, int searchval)  int mid;  if (low <= high) {  mid = (low+high)/2;  if (searchval < values[mid] return binSearch(values, low, mid-1, searchval);  else if (searchval > values[mid]) return binSearch(values, mid+1, high, searchval);  else return 1;  }  return 0;  } | | | | | | | | | | | | | | void printReverse(char word[], int length) {  if (length > 0) {  printf(“%c”, word[length-1]);  printReverse(word, length-1);  }  } | | | | | | void dectobin(int n) {  if (n < 2) printf("%d", n);  else {  dectobin(n/2);  printf("%d", n%2);  }  } | | | | | | | int modPow(int base, int exp, int n) {  //O(exp)  base = base%n;  if (exp == 0) return 1;  else if (exp == 1) return base;  else if (exp%2 == 0) return modPow(base\*base%n, exp/2, n);  else return base\*modPow(base, exp-1, n)%n;  } | | |
| void towers(int n, int start, int end) {  int mid;  if (n > 0) {  mid = 6 - start - end;  towers(n-1, start, mid);  printf("Move disk %d from tower ", n);  printf("%d to tower %d.", start, end);  towers(n-1,mid,end);  }  } | | | | | | | //1) Move the subtower of n-1 disks from pole 1 to pole 3.  //2) Move the bottom disk to pole 2.  //3) Move the subtower of n-1 disks from pole 3 to pole 2. | | | | | | | void floodfill(char grid[][SIZE], int x, int y) {  int i,j;  grid[x][y] = 'FILL CHARACTER'; // Mark this spot  // Go through all valid adjacent squares with a 'd'.  for (i=-1; i<2; i++)  for (j=-1; j<2; j++)  if (inbounds(x+i,y+j) && grid[x+i][y+j]=='d')  floodfill(grid,x+i,y+j);    } | | | | | | | | | | | | **Useful Functions** | | | |
| void print(int array[], int n) {  int i;  for (i=0; i<n; i++) printf("%d ", array[i]);  printf("\n");  }  void swap(int \*a, int \*b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  } | | | |
| **Permutations & Brute Force** | | | | | | | | | | | | | | | | | | | | | | | | | | | Merge Sort [O(*n*lg*n*)] | | |
| void runPerms() {// Prints permutations of 0,1,2,3.  int perm[4];  int i, used[4];  for (i=0; i<4; i++) used[i] = 0;  printPerms(perm, used, 0, 4);  printf("\n");  }  void printperms(int\* perm, int\* used, int k, int n) {  if (k == n) print(perm, n);  int i;  for (i=0; i<n; i++) {  if (!used[i]) {  used[i] = 1;  perm[k] = i;  printperms(perm, used, k+1, n);  used[i] = 0;  }  }  }  void nextPerm(int perm[], int length) {  // Find the spot that needs to change  int i = length-1;  while (i>0 && perm[i] < perm[i-1]) i--;  i--; // Advance to swap location.  // So last perm doesn't cause a problem.  if (i == -1) return;  // Find the spot with which to swap.  int j=length-1;  while (j>i && perm[j]<perm[i]) j--;  // Swap it.  int temp = perm[i];  perm[i] = perm[j];  perm[j] = temp;  // reverse from index i+1 to length-1.  int k,m;  for (k=i+1,m=length-1; k<m; k++,m--) {  temp = perm[k];  perm[k] = perm[m];  perm[m] = temp;  }  } | | | | | | | | void runOdometer() {// Runs a 4 digit odometer.  int meter[4];  printOdometer(meter, 0, 4);  printf("\n");  }  void printOdometer(int odometer[], int k, int n) { // Prints all possible seetings of odometer with n digits with the first k fixed.  // Base case.  if (k == n) print(odometer, n);  // Fill in each possible digit in slot k and recurse.  else {  int i;  for (i=0; i<10; i++) {  odometer[k] = i;  printOdometer(odometer, k+1, n);  }  }  } | | | | | | | | | void runDerangements() { // Prints out all derangements of 0,1,2,3,4  int perm[4];  int i, used[4];  for (i=0; i<4; i++) used[i] = 0;  printDerangements(perm, used, 0, 4);  printf("\n");  }  void printDerangements(int perm[], int used[], int k, int n) { // Prints out all derangements in perm with the first k digits fixed.  // Base case.  if (k == n) print(perm, n);  // Recursive case - fill in slot k.  else {  int i;  // Same as permutation, but we don't allow slot k to be filled with k.  for (i=0; i<n; i++) {  if (!used[i] && i != k) {  used[i] = 1;  perm[k] = i;  printDerangements(perm, used, k+1, n);  used[i] = 0;  }  }  }  }  void derange(int[] perm, boolean[] used, int k) {  if (k == perm.length) print(perm, perm.length, true);  for (int i=0; i<perm.length; i++) {  if (!used[i] && i != k) {  used[i] = true;  perm[k] = i;  derange(perm, used, k+1);  used[i] = false;  }  }  } | | | | | | | | | | 1) Keep track of the smallest value in each array that hasn’t been placed in order in the larger array yet.  2) Compare these two smallest values from each array. One of these must be the smallest of all the values in both arrays that are left. Place the smallest of the two values in the next location in the larger array.  3) Adjust the smallest value for the appropriate array.  4) Continue this process until all values have been placed in the large array.  1) Sort the first half of the array separately, using Merge Sort.  2) Sort the second half of the array separately, using Merge Sort.  3) Now, we do have a situation to use the Merge algorithm! Simply merge the first half of the array with the second half. | | |
| {2, 7, 16, 44, 55, 89}and{1, 6, 9, 13, 15, 49}    void Merge(int values[], int start, int middle, int end) {  int \*temp, i, length, count1, count2, mc;  length = end - start + 1;  temp = (int\*)calloc(length, sizeof(int));  // Indexes into our two sorted lists.  count1 = start;  count2 = middle;  // Keeps track of our index into our auxiliary array.  mc = 0;  // Copy values into our auxiliary array, so long as there are numbers from both lists to copy.  while ((count1 < middle) || (count2 <= end)) {  // Next value to copy comes from list one - make sure list 1 isn't exhausted yet. Also make sure we don't access index count2 if we aren't supposed to.  if (count2 > end || (count1 < middle && values[count1] < values[count2])) {  temp[mc] = values[count1];  count1++;  mc++;  }  // We copy the next value from list two.  else {  temp[mc] = values[count2];  count2++;  mc++;  }  }  // Copy back all values into original array.  for (i=start; i<=end; i++)  values[i] = temp[i - start];  free(temp);  }  void MergeSort(int values[], int start, int end) {  int mid;  // Check if sorting range is > one element  if (start < end) {  mid = (start+end)/2;  // Sort the first half of the values.  MergeSort(values, start, mid);    // Sort the last half of the values.  MergeSort(values, mid+1, end);    // Put it all together.  Merge(values, start, mid+1, end);  }  } | | |
| void runCombos() {  int i, items[5];  for (i=0; i<5; i++) items[i] = 0;  printCombos(items, 0, 5);  printf("\n");  }  void printCombos(int subset[], int k, int n) {  // Base case, subset filled in.  if (k == n) printSubsets(subset, n);  // Recursive case - fill slot k.  else {  // First do subset without item k.  printCombos(subset, k+1, n);  // Now do the subset with item k. Must return subset to original setting!!!  subset[k] = 1;  printCombos(subset, k+1, n);  subset[k] = 0;  }  }  void printSubsets(int subset[], int n) { // Prints out the subset of 0,1,2..,n-1 represented by subset. subset[i] is 1 iff i is in the subset.  int i;  for (i=0; i<n; i++)  if (subset[i])  printf("%d ", i);  printf("\n");  } | | | | | | | | |
| **O(*n*2) Sorts** | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Selection Sort [O(*n2*)] | | | | | | | Insertion Sort [O(*n*) - O(*n2*) - O(*n2*)] | | | | | | | | | | Bubble Sort [O(*n2*)] | | | | | | | | | |
| For the ith element (as i ranges from 0 to n-1)  1) Determine the smallest element in the rest of the array to the right of the ith element.  2) Swap the current ith element with the element identified in step 1. | | | | | | | For the ith element (as i ranges from 1 to n-1)   1. As long as the current element is greater than the previous one, swap the two elements. Stop if there's no previous element. 2. Number of steps varies by input | | | | | | | | | | Always compare consecutive elements, left to right.  Whenever two elements are out of place, swap them.  At the end of a single iteration, the maximum element will be in the last spot.  Repeat n times. On each pass, one more maximal element will be put in place. | | | | | | | | | |
| 8, 3, 1, 9, 5, 2  Determining where the smallest element is stored. Swap this with element at start to yield:  **1**, 3, 8, 9, 5, 2.  Similarly, find the next smallest and swap it:  1, **2**, 8, 9, 5, 3.  1, 2, **3**, 9, 5, 8  1, 2, 3, **5**, 9, 8  1, 2, 3, 4, **8**, 9 | | | | | | | 3, 7, 2, 1, 5  **3**, **7**, 2, 1, 5  3, **2**, **7**, 1, 5  **2**, **3**, 7, 1, 5  2, 3, **1**, **7**, 5  2, **1**, **3**, 7, 5  **1**, **2**, 3, 7, 5  1, 2, 3, 5, 7 | | | | | | | | | | 6, 2, 5, 7, 3, 8, 4, 1  2, **6**, 5, 7, 3, 8, 4, 1  2, 5, **6**, 7, 3, 8, 4, 1  2, 5, **6**, 7, 3, 8, 4, 1  2, 5, 6, 3, **7**, 8, 4, 1  2, 5, 6, 3, **7**, 8, 4, 1  2, 5, 6, 3, 7, 4, **8**, 1  2, 5, 6, 3, 7, 4, 1, **8**  Maximum element is in place; repeat process | | | | | **2**, 5, 6, 3, 7, 4, 1, 8  2, **5**, 6, 3, 7, 4, 1, 8  2, 5, 3, **6**, 7, 4, 1, 8  2, 5, 3, **6**, 7, 4, 1, 8  2, 5, 3, 6, 4, **7**, 1, 8  2, 5, 3, 6, 4, 1, **7**, 8  2, 3, **5**, 6, 4, 1, 7, 8  2, 3, **5**, 6, 4, 1, 7, 8  … | | | | |
| void selectionSort(int A[], int n) {  int cur, j, smallest;  for (cur = 0; cur <n; cur++) {  smallest = cur;  for (j=cur+1; j<n; j++)  if (A[j] < A[smallest])  smallest = j;  swap(&A[cur], &A[smallest]);  if (0) printArray(A, n);  }  printf("\n");  } | | | | | | | void insertionSort(int A[], int n) {  int i,j;  for (i=1; i<n; i++) {  j=i;  while (j > 0 && A[j] < A[j-1]) {  swap(A+j, A+j-1);  j--;  }  if (0) printArray(A, n);  }  printf("\n");  } | | | | | | | | | | void bubbleSort(int A[], int n) {  int i,j;  for (i=n-2; i>=0; i--) {  for (j=0; j<=i; j++)  if (A[j] > A[j+1]) swap(&A[j], &A[j+1]);  if (0) printArray(A, n);  }  printf("\n");  }  void RbubbleSort(int A[], int n) {  int i, j;  if (n == 1) return;  bubblesort(Array,n-1);  } | | | | | | | | | |
| **Quick Sort** [O(*n*lg*n*) - ? - O(*n2*)] | | | | | | | | | int partition(int\* vals, int low, int high) {  int temp;  int i, lowpos;  if (low == high) return low;  i = low + rand()%(high-low+1);  swap(&vals[low], &vals[i]);  lowpos = low;  low++;  while (low <= high) {  while (low <= high && vals[low] <= vals[lowpos]) low++;  while (high >= low && vals[high] > vals[lowpos]) high--;  if (low < high){  if(high – low < 40) insertionsort(array, low, high); | | | | | | | | | | | | else swap(&vals[low], &vals[high]);  }  swap(&vals[lowpos], vals[high]);  return high;  }  void quicksort(int\* numbers, int low, int high) {  if (low < high) {  int split = partition(numbers,low,high);  quicksort(numbers,low,split-1);  quicksort(numbers,split+1,high);  }  } | | | | | | | | Assume: original value = 5  **5**, 3, 6, 9, 2, 4, 7, **8**  5, **3**, 6, 9, 2, 4, **7**, 8  5, 3, **6**, 9, 2, **4**, 7, 8  5, 3, **4**, 9, 2, **6**, 7, 8  5, 3, 4, **9**, **2**, 6, 7, 8  5, 3, 4, **2**, **9**, 6, 7, 8  2, 3, 4, **5**, **9**, 6, 7, 8  return 5 |
| 1) Partition the array with respect to a random element.   * The median of 3-5 elements   2) Sort the left part of the array, using Quick Sort  3) Sort the right part of the array, using Quick Sort. | | | | | | | | |