**Introduction to Algorithms**

Characteristics

1. Space complexity – how much memory does it require
2. Time complexity – how much time does it take to complete

Common algorithms

1. Search algorithms – find specific data in a structure
2. Sorting algorithms

Data structures

* Arrays
* Linked lists
* Stacks and queues
* Trees
* Hash tables

**Bubble sort**

23, 8, 15, 17, 4, 40, 11, 31

Compare index 0 (23), to index 1 (8), if index 0 is greater than index 1 then swap values and move on to comparing index 1 with index 2.

8, 23, 15, 17, 4, 40, 11, 31

8, 15, 23, 17, 4, 40, 11, 31

Other sorting algorithms are generally more efficient

Int numbers[8] = {23, 8, 15, 17, 4, 40, 11, 31};

For (int I = 0; I < 7; i++) {

For (int j = 0; j < 7; j++) {

If (numbers[j] > numbers[j+1]) {

Int tmp = numbers[j];

Numbers[j] = numbers[j+1];

Numbers[j+1] = tmp;

}

}

}

A nested for loop potentially indicates a performance issue

**Merge sort**

* Break a data set into individual pieces and merges them
* Uses recursion
* Performs well on large datasets

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 19 | 31 | 4 | 23 |

|  |  |  |
| --- | --- | --- |
| 12 | 19 | 31 |

|  |  |
| --- | --- |
| 4 | 23 |

Imagine the two pointers (arrows) at the start of each sub-array. These 2 are compared, e.g. 12 is compared to 4 and the lowest one is merged in to the result array. The pointer on the element which was merged then moves along, the other pointer stays where it is until that element is merged in to the result array.

In the above example; 4 < 12 (the pointer in sub array 2 now goes to 23, and 4 goes to the result array). 12 < 23 so 12 goes in to the result array and moves along to now point at 19. 19 < 23, 23 < 31, 31 remaining…

4, 12, 19, 23, 31

#define max 10

Int a[11] = {10, 14, 6, 3, 8, 22, 1, 13, 0, 20, 21};

Int b[10];

Void merging (int low, int mid, int high) {

Int list1, list2, i;

For (list1 = low, list2 = mid + 1, i = low; list1 <= mid && list2 <= high; i++) {

If (a[list1] <= a[list2]) {

b[i] = a[list1++];

} else {

b[i] = a[list2++];

}

}

// if list 1 array still has values, add them

While (list1 <= mid) {

b[i++] = a[list1++];

}

// same for list 2

While(list2 <= high) {

b[i++] = a[list2++];

}

// merge back in to the original array

For (i = low; I <= high; i++) {

a[i] = b[i];

}

}

Void sort(int low, int high) {

Int mid;

If (low < high) {

Mid = (low + high) / 2;

Sort(low, mid); // first sub-array

Sort(mid + 1, high); // second sub-array (hence mid + 1 as mid incl above)

Merging(low, mid, high);

} else {

Return;

}

}

Int main() {

Sort(0, max);

Return 0;

}

**Quicksort**

Uses recursion and generally performs better than merge sort.

20, 6, 8, 53, 23, 87, 42, 19

Pivot

Pick a pivot position, now imagine 2 arrows – one after the pivot position and the other at the end. The arrow next to the pivot position checks if that element is greater than the pivot, the arrow at the end looks for a number less than the pivot. The arrows move closer together until they find what they are looking for (a number lesser than or greater than). At this point the numbers swap

20, 6, 8, 53, 23, 87, 42, 19

The right arrow is less, so it stays where it is, the left arrow began at 6, which is less than so it moves until it finds a number greater than which is 53. 53 now swaps with 19

20, 6, 8, 19, 23, 87, 42, 53

23 is greater than the pivot so the left arrow stays where it is. The right arrow has now become greater so it moves to the left until it finds a number less than the pivot. This happens at 19, which *crosses* over the other arrow. When this happens this defines the splitting point at where to split the array and the original pivot is put at this point, so we end up with:

19, 6, 8 20 23, 87, 42, 53

#include<stdio.h>

void quicksort(int number[25],int first,int last){

int i, j, pivot, temp;

if(first<last){

pivot=first;

i=first;

j=last;

while(i<j){

while(number[i]<=number[pivot]&&i<last)

i++;

while(number[j]>number[pivot])

j--;

if(i<j){

temp=number[i];

number[i]=number[j];

number[j]=temp;

}

}

temp=number[pivot];

number[pivot]=number[j];

number[j]=temp;

quicksort(number,first,j-1);

quicksort(number,j+1,last);

}

}

int main(){

int i, count, number[25];

printf("How many elements are u going to enter?: ");

scanf("%d",&count);

printf("Enter %d elements: ", count);

for(i=0;i<count;i++)

scanf("%d",&number[i]);

quicksort(number,0,count-1);

printf("Order of Sorted elements: ");

for(i=0;i<count;i++)

printf(" %d",number[i]);

return 0;

}

**Searching algorithms**

Unordered list search/sequential search

10, 50, 30, 70, 80, 20, 60 ,90, 40 // find 20

Int nums[10] = {10, 20, 30, 40, 50, 60, 70 ,80 ,90};

Int find\_index(int num, int nums[10]) {

For (int i = 0; I < 10; i++) {

If (num == nums[i]) {

Return i;

}

}

}

This can be inefficient especially on large data sets. It is much better to search in sorted data.

Binary search

6, 8, 19, 20, 23, 41, 49, 53, 96

Get the mid point – 1 (so in the list above this would be 20 (index 3). If the number we are looking for is higher than the mid point, we know we can ignore the numbers below it. From here we create a new mid point between 20 and 96 which is index 5 (value 41). At this point we assess again.

Int binary\_search(int arr[], int length, int upper, int lower, int find) {

If (upper >= lower) {

Int mid = lower + (upper – lower) / 2;

// check if element mid point is the one we are looking for

If (arr[mid] == find) {

Return mid;

}

// if element is smaller than mid, then it can only be present in left sub-array

If (arr[mid] > find) {

Return binary\_search(arr, length, mid – 1, lower, find);

}

// else must be present in right sub array

Return binary\_search(arr, length, upper, mid + 1, find);

}

Return -1; //if *find* not found

}

/\***side note** – linear time complexity refers to the time to calculate increases as the input increases such as with a linear search when we traverse each element individually. Linear time complexity AKA O(n) (where O is big-O notation) \*/

Check if a list is sorted

int is\_sorted(int arr[]) {

int i;

for (i = 0; i < //lengthOfArray - 1; i++) {

if (arr[i] > arr[i+1]) {

return FALSE;

}

}

Return TRUE;

}