Link:- <https://www.youtube.com/watch?v=RpRRUQFbePU>

Don’t senselessly try and try again, work **systematically**.

Preconditions:- A Condition that must be satisfied before entering a function

Use assert to determine whether Preconditions are met.

Post conditions:- The qualities you can rely on after the function is completed.

**Computing:-**

P:- Problem, A:- Algorithm, P:- Program, P:- Process (run program)

Brain is the most important asset – keep it simulated during lecture by asking questions and thinking of new problems, that way you learn better. Also take a walk during the breaks.

Science -> Make hypothesis then try proving it wrong -> if no one can prove it wrong then the hypothesis may be correct. Provided the hypothesis can be tested.

Algorithm:-

1. Selection Sort:-
   1. Time Complexity – Worst, Best and Average – O(n^2)
   2. Algorithm:- Keep finding minimum from list

*/\* a[0] to a[n-1] is the array to sort \*/*

int i,j;

int iMin;

*/\* advance the position through the entire array \*/*

*/\* (could do j < n-1 because single element is also min element) \*/*

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

*/\* find the min element in the unsorted a[j .. n-1] \*/*

*/\* assume the min is the first element \*/*

iMin = j;

*/\* test against elements after j to find the smallest \*/*

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

*/\* if this element is less, then it is the new minimum \*/*

if (a[i] < a[iMin]) {

*/\* found new minimum; remember its index \*/*

iMin = i;

}

}

if(iMin != j) {

swap(a[j], a[iMin]);

}

}

1. Bubble Sort:-
   1. Time Complexity – Worst and Average – O(n^2)

Best - O(n)

* 1. Algorithm:- Keep pushing the minimum or maximum element to one end of the list
  2. Variation:- cocktailSort -> Where we alternately push the max element to one end and the min element to the other.
  3. Approx number of steps -> n(n-1)/2

**procedure** cocktailSort( A **:** list of sortable items ) **defined as:**

**do**

swapped := false

**for each** i **in** 0 **to** length( A ) - 2 **do:**

**if** A[ i ] > A[ i + 1 ] **then** // test whether the two elements are in the wrong order

swap( A[ i ], A[ i + 1 ] ) // let the two elements change places

swapped := true

**end if**

**end for**

**if** swapped = false **then**

// we can exit the outer loop here if no swaps occurred.

**break do-while loop**

**end if**

swapped := false

**for each** i **in** length( A ) - 2 **to** 0 **do:**

**if** A[ i ] > A[ i + 1 ] **then**

swap( A[ i ], A[ i + 1 ] )

swapped := true

**end if**

**end for**

**while** swapped // if no elements have been swapped, then the list is sorted

**end procedure**

1. Insertion Sort

Time Complexity – Average and Worst - O(n^2)

Best case it is O(n)

Algorithm:- For every element make sure that the elements before it are sorted.

For the 0th element all elements before it are srted hence you can start directly from 1st element.

**for** i = 1 **to** length(A) - 1

x = A[i]

j = i

**while** j > 0 and A[j-1] > x

A[j] = A[j-1]

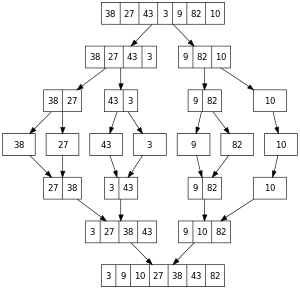
j = j - 1

A[j] = x

1. Merge Sort:-

|  |  |
| --- | --- |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) typical,  O(*n*) natural variant |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) |

Algorithm:- Keep dividing the problem into subproblems and then merge the result.



TopDownMergeSort(A[], B[], n)

{

TopDownSplitMerge(A, 0, n, B);

}

*// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set)*

TopDownSplitMerge(A[], iBegin, iEnd, B[])

{

if(iEnd - iBegin < 2) *// if run size == 1*

return; *// consider it sorted*

*// recursively split runs into two halves until run size == 1,*

*// then merge them and return back up the call chain*

iMiddle = (iEnd + iBegin) / 2; *// iMiddle = mid point*

TopDownSplitMerge(A, iBegin, iMiddle, B); *// split / merge left half*

TopDownSplitMerge(A, iMiddle, iEnd, B); *// split / merge right half*

TopDownMerge(A, iBegin, iMiddle, iEnd, B); *// merge the two half runs*

CopyArray(B, iBegin, iEnd, A); *// copy the merged runs back to A*

}

*// left half is A[iBegin :iMiddle-1]*

*// right half is A[iMiddle:iEnd-1 ]*

TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])

{

i0 = iBegin, i1 = iMiddle;

*// While there are elements in the left or right runs*

for (j = iBegin; j < iEnd; j++) {

*// If left run head exists and is <= existing right run head.*

if (i0 < iMiddle && (i1 >= iEnd || A[i0] <= A[i1]))

B[j] = A[i0];

i0 = i0 + 1;

else

B[j] = A[i1];

i1 = i1 + 1;

}

}

CopyArray(B[], iBegin, iEnd, A[])

{

for(k = iBegin; k < iEnd; k++)

A[k] = B[k];

}

1. Shell sort

|  |  |
| --- | --- |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n*2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | depends on gap sequence |
|  |  |

Algorithm:- Calculate gaps in the sequence using various techniques then use insertion/buble sort on

Gap1 -> End then Gap2 -> End then Gap 3 -> End

Where Gap1 > Gap 2 > Gap 3

*# Sort an array a[0...n-1].*

gaps = [701, 301, 132, 57, 23, 10, 4, 1]

*# Start with the largest gap and work down to a gap of 1*

[**foreach**](http://en.wikipedia.org/wiki/Foreach) (gap in gaps)

{

*# Do a gapped insertion sort for this gap size.*

*# The first gap elements a[0..gap-1] are already in gapped order*

*# keep adding one more element until the entire array is gap sorted*

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

{

*# add a[i] to the elements that have been gap sorted*

*# save a[i] in temp and make a hole at position i*

temp = a[i]

*# shift earlier gap-sorted elements up until the correct location for a[i] is found*

**for** (j = i; j >= gap and a[j - gap] > temp; j -= gap)

{

a[j] = a[j - gap]

}

*# put temp (the original a[i]) in its correct location*

a[j] = temp

}

}

Where Gaps are

\left\lfloor\frac{N}{2}\right\rfloor,
        \left\lfloor\frac{N}{4}\right\rfloor, \ldots, 1

1. Quick sort

The steps are:

1. Pick an element, called a **pivot**, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. [Recursively](http://en.wikipedia.org/wiki/Recursion_(computer_science)) apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The [base case](http://en.wikipedia.org/wiki/Recursion_(computer_science)) of the recursion is arrays of size zero or one, which never need to be sorted. In [pseudocode](http://en.wikipedia.org/wiki/Pseudocode), a quicksort that sorts elements *i* through *k* (inclusive) of an array *A* can be expressed compactly as[[5]](http://en.wikipedia.org/wiki/Quicksort#cite_note-5):171

quicksort(A, lo, hi):

**if** lo < hi:

p := partition(A, lo, hi)

quicksort(A, lo, p - 1)

quicksort(A, p + 1, hi)

*// lo is the index of the leftmost element of the subarray*

*// hi is the index of the rightmost element of the subarray (inclusive)*

partition(A, lo, hi)

pivotIndex := choosePivot(A, lo, hi)

pivotValue := A[pivotIndex]

*// put the chosen pivot at A[hi]*

swap A[pivotIndex] and A[hi]

storeIndex := lo

*// Compare remaining array elements against pivotValue = A[hi]*

**for** i **from** lo **to** hi−1, **inclusive**

**if** A[i] < pivotValue

swap A[i] and A[storeIndex]

storeIndex := storeIndex + 1

swap A[storeIndex] and A[hi] // *Move pivot to its final place*

**return** storeIndex

Any value can be chosen as pivot and separate arguments are there for each choice

1. HeapSort

|  |  |
| --- | --- |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | \Omega(n), O(n\log n)[[1]](http://en.wikipedia.org/wiki/Heapsort#cite_note-1) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n) |

Algorithm:- Put the entire structure in a heap and then keep extracting the root node.

Comparison based on:-

1. Time taken – no of cycles
2. Space required

Provided run on same PC with same CPU and memory availability

**Optimization Priorities:-**

1. Problem -> Relevant only to jobs i.e try to remove unnecessary complexities which are not very important to end users.
2. Algorithm -> Picking the right algorithm is the deciding factor.
3. Programming language
4. Stack vs Queue
5. Comments in case of interpreted languages
6. couts and file i/o’s etc…

**Properties of an algorithm:-**

1. Linearity -> If I double the number of inputs does the time taken doubles?
   1. Quadratic -> Square function -> 2 inputs
   2. Cubic -> need three different number of inputs to calculate if it’s cubic
   3. Polynomial -> requires (N+1) number of measurements

Important:-

1. Never fall into the trap of thinking that measuring the time it takes for a program to run tells you something useful about the speed of the algorithm. Absolute speed is irrelevant, algorithms are by how the speed changes between 2 different sized inputs i.e Relative speed. Changing the size of the input is not enough try changing the number of binary bits it takes to represent the input also.

Example:- Finding if a number is prime:-

Run time -> N-1 -> i.e see if the no is divisible by 1 … N-1

Hence for 7->6, 8->7, 9->8 but this algo is exponential (2^n) and not linear since for 3 bit number (7) it takes (2^3) time for 15 ->4 bit number it takes (2^4) time

1. Good Algos -> Linear time or logarithmic time
2. Mediocre -> (n^2), (n^3) etc
3. Bad -> Exponential

**Puzzles:-**

1. We have 4 identical people 2 are from Mars and 2 from Venus, one from each planet always lies and one always speaks the truth.

What Q can I ask to find which planet the people are from .

Ans:- Will your partner say you are from Mars? The people who say No are from Mars the people who say yes are from Venus.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Planet | Mars | Mars | Venus | Venus |
| T/F Person | T | F | T | F |
| Partners Answer | No | Yes | Yes | No |
| Reported Answer | No | No | Yes | Yes |
| Conclusion | Mars | Mars | Venus | Venus |
| Question :- Will your partner say you are from Mars? The people who say No are from Mars the people who say yes are from Venus | | | | |

1. Using only one Q can we find which person is from Mars and speaks the Truth and which doesn’t -> Same from Venus.

Ans:- Using only one question this can’t be determined.

Theory of computation states that for representing N States I need at least

* ceil(log (N - 1)) -> Base 2 Binary bit
* ceil(log (N - 1)) -> Base 4 Hexa numbers

In our case Yes or No is a Binary Bit and one Q means one Binary bit -> hence we can represent only 2 states i.e Mars or Venus. To determine True or False we need another bit.

**Maths**

1. E(n) i.e summation of n = 1 + 2 + 3 …n = n(n+1)/2

Lessons Learnt:-

1. Stable sort means that the elements other than the required will not be re ordered.

Eg:- If I have a set of points already sorted according to Y co ordinate and then I sort it according to X co ordinate then I automatically get a set of points in X then Y co ordinate. i.e for points having the same X co ordinate the points are co ordinate sorted according to Y Co ordinate.

For example:  
before: (x,y)= (2,2),(2,3),(1,2),(1,3),(2,1),(1,1),(3,2),(3,3),(3,1)  
sorted by y: (x,y)= (2,1),(1,1),(3,1),(2,2),(1,2),(3,2),(2,3),(1,3),(3,3)  
sorted by x: (x,y)= (1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)

1. Binary search can’t work in O(log n) on linked list because binary search requires O(1) to access any element (specially the mid element).