What is a brute force algorithm and how do you discover them?

↳ a straightforward approach to solving an algorithm, usually not fast, but very applicable. You find it by simply doing the problem as stated and nothing more.

3.1 Selection Sort

A.) How does selection sort improve in bubble sort?

Selection sort improves bubble sort by requiring one less swap.

While Bubble is always $O(n^2)$ swaps, selection can do $O(n)$

B.) What is an in-place sort / what is a stable sort?

- **In-place**: A sorting algorithm in which sorted items occupy the same storage as the original ones
  examples: Bubble sort, selection sort, insertion sort, heap sort ~~merge sort~~
- **Stable**: If two objects with equal keys appear in the same order in the sorted output as they are in the input array to be sorted.

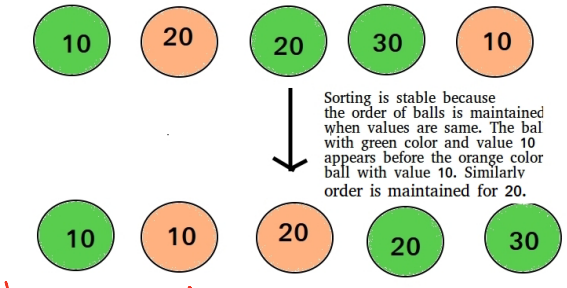
- Insertion Sort
- Merge Sort
- Bubble Sort
- Tim Sort
- Counting Sort
- Block Sort
- Quadsort
- Library Sort
- Cocktail shaker Sort
- Gnome Sort
- Odd–even Sort

Stable

Unstable Sorting Algorithms:

- Heap sort
- Selection sort
- Shell sort
- Quick sort
- Introsort (subject to Quicksort)
- Tree sort
- Cycle sort
- Smoothsort
- Tournament sort(subject to Hesapsort)

not stable



The orange 10 still comes after the green 10.

3.2 Sequential Search

↳ The sequential sort, while inefficient, is very simple to understand and implement.

↳ All it does is search each element successively until it finds a match.

↳ $O(n)$ time; $O(1)$ space complexities.

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in $A[0..n-1]$ whose value is
// equal to K or -1 if no such element is found
$A[n] \leftarrow K$
$i \leftarrow 0$
while $A[i] \neq K$ **do**
  $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

4?

	best	worst	avg.
Item is present	1	n	n/2
item not present	n	n	n

3.3 Closest pairs

↳ Tasked with finding the two closest points in a set of n points

↳ Uses the standard Euclidian distance formula:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

↳ **ALGORITHM** *BruteForceClosestPair*(P)

//Finds distance between two closest points in the plane by brute force
//Input: A list P of n $(n \geq 2)$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
return d

We can improve this by not taking the square root. Big numbers have big square roots and small numbers have small roots.

Therefore... the basic operation is squaring a number!

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i)$$
$$= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2).$$

3.4 Exhaustive Search

↳ A brute force approach to combinatorial problems

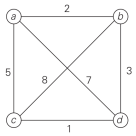
↳ Suggests generating each and every element of the problem domain, selecting those that satisfy the constraints and then finding a desired element. That is why it is brute force. Looks at *everything*

Traveling Salesman Problem.

↳ Find the shortest tour through a given set of cities n that visits each once before returning to the original city.

↳ Possible solutions?

- Use a weighted graph and find the shortest Hamiltonian Circuit.
- or define it as a sequence of n+1 adjacent vertices.



Tour	Length	
a —> b —> c —> d —> a	l = 2 + 8 + 1 + 7 = 18	
a —> b —> d —> c —> a	l = 2 + 3 + 1 + 5 = 11	optimal
a —> c —> b —> d —> a	l = 5 + 8 + 3 + 7 = 23	
a —> c —> d —> b —> a	l = 5 + 1 + 3 + 2 = 11	optimal
a —> d —> b —> c —> a	l = 7 + 3 + 8 + 5 = 23	
a —> d —> c —> b —> a	l = 7 + 1 + 8 + 2 = 18	

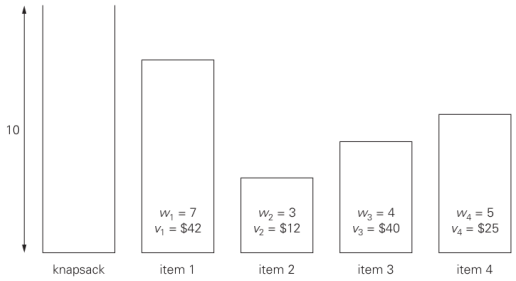
Total number of permutations is still

$\frac{1}{2}(n-1)!$ → only small instances.

Large values takes way too long.

0 knap Sack Problem

of bag capacity W and items with w and v (weight and value), what is the most valuable filled bag?



(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

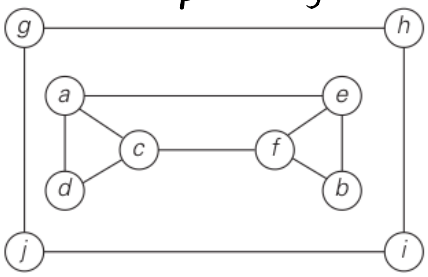
- Possible solution: generate all subsets of n items given. Compute total weight for applicable subsets, and finding the subset that has the largest value.
- Size is very restrictive as this algorithm is 2^n

NP-HARD - no polynomial time algorithm is known, and scientists think they are impossible to find an algorithm.

3.5 Depth First Search

DFS: Algorithm proceeds to the next unvisited vertex that is adjacent to the current vertex.

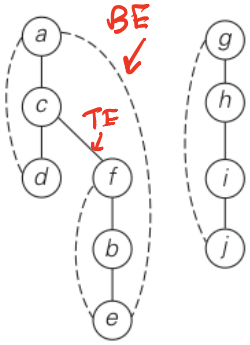
- Dead end: a vertex with no adjacent unvisited vertices.
- DFS Forest: Starting vertex is ~~root~~, new unvisited vertex is a child of the current vertex and is connected by a tree edge.
- Back Edge: This happens when an encountered edge leads to a previously visited vertex.



(a)

$e_{6,2}$
$b_{5,3}$ $j_{10,7}$
$d_{3,1}$ $f_{4,4}$ $i_{9,8}$
$c_{2,5}$ $h_{8,9}$
$a_{1,6}$ $g_{7,10}$

(b)



(c)

ALGORITHM *DFS(G)*

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)
```

```
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable count
count ← count + 1;  mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

Traversal time $\Theta(|V|)^2$

$\Theta(V+E)$

vertex, edge

- Tells us whether our graph is connected by seeing if all the vertexes have been visited when the program halts.
- Also checks for a cycle in the graph by looking at its backedges.

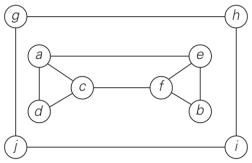
Breadth First Search!

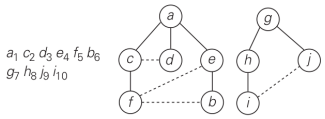
↳ Concentric manner in which you visit all that are adjacent first before moving on.

↳ uses a queue where the starting vertex gets the first spot in the queue.

BFS Tree: Traversal's starting vertex is the root, and connected with a tree edge to subsequent unvisited vertex's.

Cross edge: An edge that leads to a previously visited vertex





- $O(v^2)$ time complexity.
- Yields a single ordering of vertices due to FIFO of queue.
- Cannot find articulation points like DFS.
- Can be used to find path w/ the fewest number of edges.

ALGORITHM *BFS*(G)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable count
count ← count + 1;  mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;  mark w with count
            add w to the queue
    remove the front vertex from the queue
```

DFS vs. BFS

TABLE 3.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Chapter 4, Decrease and Conquer

↳ solving a problem by solving smaller instances of that problem.

↳ Incremental approach: Bottom-up approach where you iteratively start w/ the smallest instance of the problem.

↳ Decrease by a Constant: Size of the instance is reduced by one each time.

↳ Top-down : use the recursive definition

↳ Bottom up : multiply 1 by a n times.

↳ Decrease by Constant Factor: Divide instance each time by 2.

↳ not the same as Divide and conquer

divide problem into smaller subproblems that are smaller instances of the same problem.

↳ Constant factor : extends the solution.

4.1 Insertion Sort

↳ scan the elements of an array from right to left until the first element smaller than or equal to A[n-1] is found.

ALGORITHM *InsertionSort*$(A[0..n-1])$

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow A[i]$
  $j \leftarrow i-1$
  while $j \geq 0$ **and** $A[j] > v$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
  $A[j+1] \leftarrow v$

89 |	**45**	68	90	29	34	17
45	89 |	**68**	90	29	34	17
45	68	89 |	**90**	29	34	17
45	68	89	90 |	**29**	34	17
29	45	68	89	90 |	**34**	17
29	34	45	68	89	90 |	**17**
17	29	34	45	68	89	90

In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

discussion of quicksort in Chapter 5.) Thus, for sorted arrays, the number of key comparisons is

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n).$$

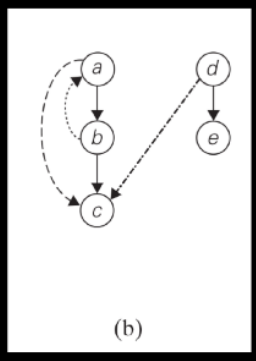
Average case for Insertion is $\frac{n^2}{4} \in \Theta(n^2)$

4.2 Topological Sort

4.2 Topological Sorting

↳ Important for directed graphs

• Direct graph: a graph specified for all its edges.



(b)

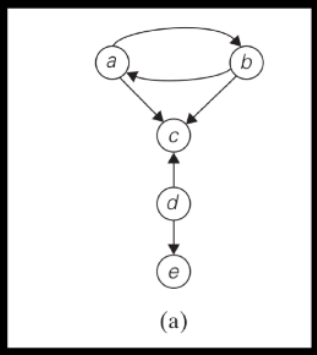
Tree edges: (ab, bc, de)

Back edges: (ba)

Forward edges: (ac)

Cross edges (dc)

Day: if a DFS forest of a digraph has no back edges (Directed acyclic graph)



(a)

Directed cycle: sequence in which three or more of a digraphs vertices that starts and ends w/ the same vertex and in which every vertex is connected to its immediate predecessor by an edge from the predecessor to the successor. (a b a)

* If a digraph has no directed cycles, the topological sorting problem for it has a solution.
* If the DFS finds a back edge, then the topological sort for the vertices is impossible.

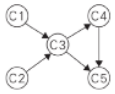
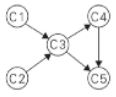


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.



(a)

$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

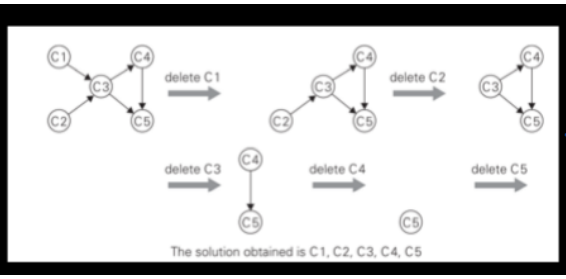
(b)

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:
C2 C1→C3→C4→C5

(c)

FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

→ DFS for topological sort.



- Source removal approach.
- Source: a vertex w/ no incoming edges.
- The order in which the vertices are removed is the solution to the topological sorting algorithm.

Example of DFS: See homework 4 page 3.

Chapter 5 Divide and Conquer

Very useful way of dividing a problem into two smaller parts and then adding those two smaller parts to find the sum of the original part.

- Ideally suited for parallel computations in which each subproblem can be simultaneously solved by its own processor.

☐ General Divide-and-conquer recurrence: $T(n) = aT(n/b) + f(n)$

Master Theorem:

$$T(n) \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Example:

$A(n) = 2A(n/2) + 1$

$a = 2;\ b = 2;\ d = 0$ since $a > b^d$

$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$

$T(n) = 4T(n/2) + n^2 \quad T(1) = 1$

$a = 4 \quad b = 2 \quad d = 2$

$4 = 2^2 \Rightarrow a = b^d$ $(n^d \log n)$

$n^2 \log n$

5.1 Mergesort

↳ Sorts a given array by dividing it into two halves and sorting those halves recursively.

↳ Then you call the merge function to merge the two together.

ALGORITHM *Mergesort*$(A[0..n-1])$
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A) //see below

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$
//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
if $B[i] \le C[j]$
$A[k] \leftarrow B[i];\ i \leftarrow i+1$
else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
$k \leftarrow k+1$
if $i = p$
copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Mergesort is: $C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$, $C(1) = 0$

$a = 2 \quad b = 2 \quad d = 0 \Rightarrow \quad a > b^d$

$\Theta(n^{\log_b a}) = \Theta(n)$

Topological Sort DFS:

- Pop off the vertexes that do not have any outgoing edges.
- Only way this is possible is if the digraph is a dag (where it has no back edges)

- Topological sort Decrease by one and conquer
  -> delete a source and its edges and move to the next one.
  -> If there are no sources, stop the problem as it cannot be solved.

Permutations:

$$P(n,r) = \frac{n!}{(n-r)!}$$

Order matters

Combinations

$$C(n,r) = \frac{n!}{(n-r)!\,r!}$$