Dynamic Programming

- dynamic programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and
- utilizing the fact that the optimal solution to the overall problem depends on the optimal solutions to its subproblems

Two Key Concepts:

1. **Optimal Substructure:**
   - A problem exhibits an optimal substructure if an optimal solution can be constructed from the optimal solutions of its subproblems.
   - This means that solving the smaller subproblems and combining their solutions yields an optimal solution to the entire problem.

   Example:

   - In the **Shortest Path Problem**, the shortest path between two nodes AA and BB can be constructed from the shortest paths between intermediate nodes along the way, thereby showing an optimal substructure.
2. **Overlapping Subproblems:**
   - A problem exhibits overlapping subproblems if the same subproblems are solved multiple times in the process of solving the main problem.
   - DP leverages this by storing solutions to subproblems to avoid redundant calculations.

   Example:

   - In the **Fibonacci Sequence Problem**, the nth Fibonacci number can be expressed as the sum of the (n-1)th and (n-2)th Fibonacci numbers, and these subproblems are repeatedly called when computing larger Fibonacci numbers.

Proving Optimal Substructure:

1. **Divide and Conquer Approach:**
   - Break the problem down into smaller subproblems.
   - Show how the optimal solution to these smaller subproblems can be combined to form the optimal solution to the entire problem.
2. **Recursive Solution:**
   - Define a recursive function that calculates the solution by combining results from smaller subproblems.
   - For example, in the **Longest Common Subsequence Problem**, the function $LCS(X,Y)L**CS(X,Y)$ can recursively call $LCS(X-1,Y)L**CS(X-1,Y)$ and $LCS(X,Y-1)L**CS(X,Y-1)$ to build the solution.
3. **Correctness Proof:**
   - Show that the recursive solution forms an optimal solution by verifying that the problem's global optimum is achieved by combining the local optima.

DP in Action:

1. **Top-Down Approach:**
   - Use recursion with memoization to store the results of subproblems, preventing repeated computations.
2. **Bottom-Up Approach:**
   - Build up the solution iteratively by solving the smallest subproblems first and storing their results in a table or array.
3. **Time Complexity:**
   - DP algorithms are often efficient with time complexities ranging from $O(n)O(n)$ to $O(n2)O(n2)$ depending on the problem, due to the reuse of subproblem solutions.

Common Examples:

1. **Knapsack Problem:**
   - Has an optimal substructure where the maximum value obtainable for a given capacity is the maximum of including or excluding the current item, plus the values obtained from the remaining capacity.
2. **Edit Distance Problem:**
   - Computes the minimum number of edits required to transform one string into another. This can be built recursively by considering the costs associated with inserting, deleting, or substituting characters.

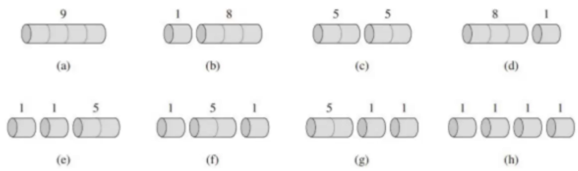
Rod Cutting

Design

- we have rod of length $n = 4$
- rods sell for different prices depending on their length. For example

length i	1	2	3	4
price p_i	1	5	8	9

- what is the optimal way we can cut up our rod to get the most revenue? Here are some example ways to cut our rod



- the number of potential cuts for a rod of length n is $2^{n-1} = 2^{4-1} = 8$
- this is exponential so testing every solution is not feasible
- assume the optimal solution cuts the rod into k pieces where $1 \leq k \leq n$
  - the optimal decomposition is $n = i_1 + i_2 + ... + i_k$
  - the corresponding optimal revenue is $r_n = p_1 + p_2 ... + p_k$

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- find the optimal revenue r for each subproblem
  - $r_1 = \max(1) = 1$
  - $r_2 = \max(2, 5) = 5$
    - where you can cut the rod into $[1, 1]$ for price $1 + 1 = 2$ or
    - $[2]$ for 5
  - $r_3 = \max(3, 6, 6, 8) = 8$
    - $[1, 1, 1]$ for $1 + 1 + 1 = 3$
    - $[1, 2]$ for $1 + 5 = 6$
    - $[2, 1]$ for $5 + 1 = 6$
    - $[3]$ for 8
  - notice we can actually reuse previous overlapping solutions
  - $r_4 = \max(9, 9, 19, 8) = 10$ where can reuse previous solutions by
    - $0 + p_4 = 9$
    - $r_1 + p_3 = 1 + 8 = 9$
    - $r_2 + p_2 = 5 + 5 = 10$
    - $r_3 + p_1 = 8 + 1 = 9$
  - $r_5 = \max(10, 13, 13, 11) = 13$
    - $0 + p_5 = 10$

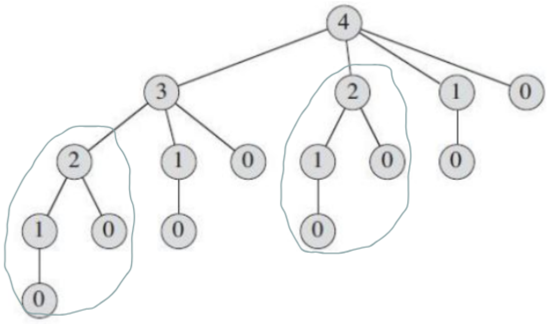
- $0 + p_5 = 10$
- $r_1 + p_4 = 1 + 9 = 10$
- $r_2 + p_3 = 5 + 8 = 13$
- $r_3 + p_2 = 8 + 5 = 13$
- $r_4 + p_1 = 10 + 1 = 11$

Without Memoization

Code

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

Runtime Analysis



- our recursion tree is made by taking initial cuts (the nodes) then recursing in `cutRod` to find the next maximum
- notice the redundancies in the tree

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$
$$T(0) = 1$$
$$T(1) = 1 + 1 = 2$$
$$T(2) = 1 + T(0) + T(1) = 4$$
$$T(3) = 1 + T(0) + T(1) + T(2) = 8$$
$$T(n) = 2^n$$

- note that you can use an inductive prove to show this
- we can use **dynamic programming** which uses additional memory to save previous computations
- there are 2 equivalent ways to reduce the repeated computation:
  - **top down**
  - **bottom up**

Top Down

We write the procedure recursively in a natural manner, but modified to save results of subproblems.

Code

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = -∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

Bottom Up

When solving a particular subproblem, we have already solved all of the smaller subproblems its solutions depends on and have those solutions saved.

Code

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

Longest Common String

- we have 2 sequences $X : (x_1, x_2, \ldots, x_m)$ and $Y : (y_1, y_2, \ldots, y_n)$
- we wish to find the longest common subsequence of X, Y
    - a subsequence of a sequence X is any sequence that can be obtained by deleting zero or more elements from X without changing the order of the remaining elements
- consider each subsequence of X corresponding to a subset of the indices $(1, 2, \ldots, m)$
    - to make a subsequence of X, you can think of it as having the option to include x or not
    - this (binary choice) yields 2^m possible subsequences
- let $Z : (z_1, z_2, \ldots, z_k)$ be any LCS of X, Y
    1. if $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
    2. if $x_m \neq y_n$ then $z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y
    3. also, if $x_m \neq y_n$ then $z_k \neq y_n$ implies Z is an LCS of X and Y_{n-1}

Recursive Design

- the conclusion we get from the 3 points above is that to find the LCS of X, Y
    - if $x_n = y_m$ then we'll find the LCS of X_{m-1}, Y_{n-1} and then append the value to it
    - otherwise we need to solve 2 subproblems
        - find the LCS of X_{m-1}, Y *and* the LCS of X, Y_{n-1} and then take the longer of these two as the LCS of X, Y
        - this is from the *implications* of the points 2 and 3 from above

Code

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

- this code uses a *bottom up* approach to DP
- initialize 2 tables b, c of size $m \times n$
    - i.e. $c(i, j)$ will hold the LCS for X_i and Y_j
- we initialize the first *row* and *column* of c with 0 as the LCS of any empty string with any other string will be length 0

- because of the bottom up structure, instead of starting from the last indices of both X, Y, we start from the first
  - the first conditional $x_i = y_i$ indicates a symbol $\nwarrow$ meaning we have a match so "cut both"
  - the second $\uparrow$ indicates cut x
  - the third $\leftarrow$ indicates cut y

Greedy Algorithms

- greedy approach considers the local optimal solutions and assumes they will lead to the global optimal solution
- this approach doesn't always work

Trying greedy approach on Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- say we have a rod of length 4
- if we try to take the greedy choice of *unit price* (i.e. price per length) we have the following units prices
  1. for 1 (cut off 1)
  2. for 2.5 (cut off 2)
  3. for 2.6 (cut off 3)
  4. for 2.25 (cut off 4)
- the greedy choice would be the sell lengths 3 then 1 for a price of 9
- however, you can sell for 2 then 2 for a price of 10
- thus the greedy solution from unit price is *not* optimal

When does greedy work?

- a greedy algorithm is a special case of DP
- in DP, bottom up approach has to consider the solutions of its children before solving itself
- however, in greedy algorithms, the solutions of children *don't* affect the current choice
  - this means *bottom up approach is invalid*

Activity Selection

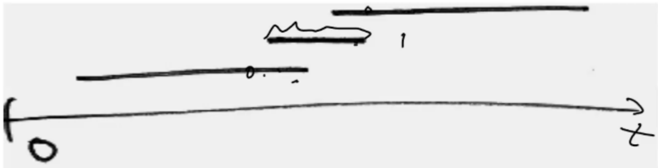
- we have a set S of n activities each with start times s_i and finish times f_i
- we'd like to schedule the maximum set of non-overlapping activities

Brute Force Approach

- we could try *all* compatible meeting combinations
- for each meeting, we'd need to chose to schedule it or not
- this gives a binary choice and a total combinations of 2^n

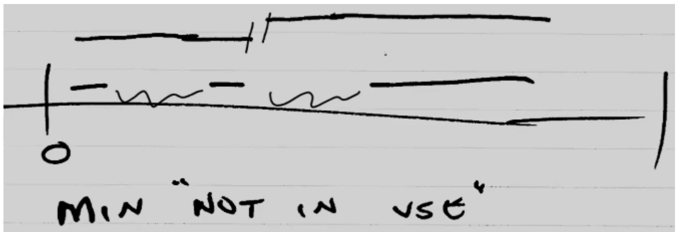
Possible Greedy Approaches

Pick the shortest meeting first



The above depicts a counter example.

Minimize "not in use" time between meetings



The above depicts a counter example.

Pick meetings with the least number of conflicts

```
           |__A.__|
|_B_||___C___||___D___||_E_|
|_____F____|     |__.__G____|
|___H____|          |___I___|
|__J_._|              |__K__|
|_L_|                   |_M_|
```

The above depicts a counter example. A has the least number of conflicts (2) while the rest have at least 3. If we pick A, at most we can schedule is 3 meetings however, we could schedule B, C, D, E for 4.

Pick the earliest start time first

The above depicts a counter example.

Pick the earliest finish time first

This greedy choice will actual give a global optimal solution to our problem. However how can we prove this?

Proving our greedy choice

- say that an optimal solution to the problem S is A
- assume that A *does not* have the greedy choice of earliest finish time in S
- take the meeting a with the earliest finish time in A
- because a is not the earliest finish time in S, there exists a meeting s in S that is not in A that has an earlier finish time
- thus, we can replace a with s with no overlap giving us a new optimal solution that *has* the greedy choice
- therefore, there is *always* an optimal solution for this problem that contains the greedy choice

0-1 Knapsack

- a thief is robbing a store with n items
- each item i is worth z_i dollars and has weight w_i where z_i, w_i are integers
- the thief can only carry W weight and can't take fractional amounts of items (i.e. 0-1 or "leave" or "take")
- what items should the thief take to maximize his haul's value?

Trying Greedy Choice

Using a greedy algorithm will not work for 0-1 knapsack. You'll need DP.

Max value first

This has easy counter examples.

Min weight first

This has easy counter examples.

Max value per weight first

- this is a.k.a. the unit price
- this has easy counter examples
- however, the *fractional* knapsack problem can be solved using this greedy choice

Proving the greedy unit price choice works for *fractional* knapsack

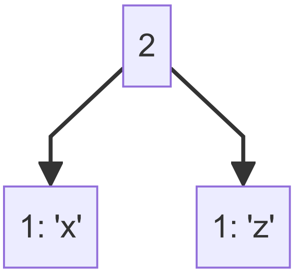
- assume we have an optimal solution A to S that doesn't have our greedy choice
- take a that has the maximum unit price in A and replace it with s that has the maximum unit price in S
- this gives us 3 possible cases:
  1. s's weight is equal to a's weight in A giving a solution with a greater value
  2. s's weight is less allowing us to fill the rest of missing weight with a fraction of a's again giving a solution with a greater value
  3. s's weight is more allowing us to take a fraction of s's again giving a solution with a greater value
- all cases lead to a contradiction that A is an optimal solution

Huffman Coding

Huffman encoding is a data compression algorithm that uses a greedy approach.

Design

1. count all the frequencies of each character and put them into an descending ordered list
2. take the bottom 2 least frequent characters and link them together by the sum of their frequencies and put that at the top of the list



3. repeat this until you have a finished tree
4. to encode a character, use `0` to denote traverse left on the tree and `1` to denote traverse right
   - once you reach a leaf (i.e. a character) you terminate
   - the sequence of `0` and `1` is the encoding

Code

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
    do allocate a new node z
       left[z] ← x ← EXTRACT-MIN(Q)
       right[z] ← y ← EXTRACT-MIN(Q)
       f[z] ← f[x] + f[y]
       INSERT(Q, z)
return EXTRACT-MIN(Q)                ▷ Return the root of the tree.
```

Amortized Analysis

- amortized analysis is the evaluation of the average cost over a sequence of operations on a data structure
- the *average cost* maybe small although a single operation can be expensive
- it is *not* the cost for the *average case* and doesn't involve probability analysis

1. Aggregate Analysis

- the amortized cost is $T(n)/n$ where $T(n)$ is the worst case
- it applies to any operation in a sequence of n operations
  - operations can be different types

Insertion to a dynamic array

- items can be inserted at a given index with $O(1)$ if the index is present in the array
- if not, then the array double in size and the cost is not longer constant

$$c_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is power of } 2 \\ 0 & \text{otherwise} \end{cases}$$

- if we insert n elements then

$$\frac{\sum_{i=1}^{n} c_i}{n} \leq \frac{n + \sum_{j=1}^{\lfloor \lg{(n-1)} \rfloor} 2^j}{n} = \frac{O(n)}{n}$$

- notice that

$$\sum_{j=0}^{a} 2^j = 2^0 + ... + 2^a = 2^{a+1} - 1$$
$$\sum_{j=1}^{\lfloor \lg{(n-1)} \rfloor} 2^j = 2^{\lfloor \lg{(n-1)} \rfloor + 1} - 1 - 1$$

- note that we subtract with another -1 because we start at $j = 1$

$$\begin{aligned} \sum_{j=1}^{\lfloor \lg{(n-1)} \rfloor} 2^j &= 2 * 2^{\lfloor \lg{(n-1)} \rfloor} - 2 \\ &= 2 * (n-1) - 2 \\ &= O(n) \end{aligned}$$

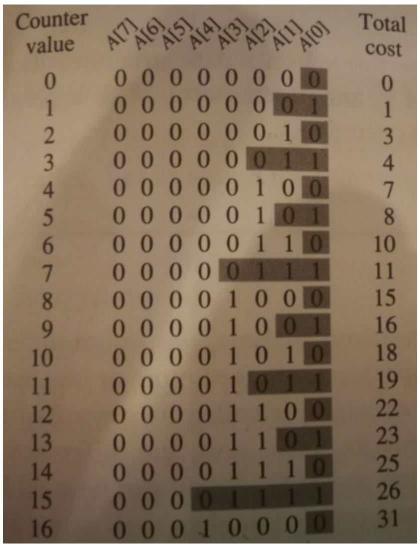
Stack Operations (`multipop`)

- `push(S, x)` pushes `x` onto `S` in $O(1)$
- `pop(S, x)` in $O(1)$
- `multipop(S, k)` pops `k` top elements from `S` if the size $\geq k$ otherwise it pops all elements
  - at most $O(n)$
- what is the amortized cost of a sequence of n `push`, `pop`, `multipop` operations?
  - the size of the stack is n

- for any n, the cost of a sequence of n of these operations is $O(n)$ (as we can't pop more than n)
- amortized cost is $O(n)/n = O(1)$

Binary Counter

- $A[0..k-1]$ is an array denoting a k-bit binary counter that starts at 0
    - adding 1 to $A[i]$ flips it
    - if $A[i] = 1$ then it yields a carry to $A[i+1]$



Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

- notice that $A[0]$ flips n times
    - $A[1]$ flips $n/2$ times
    - $A[2]$ flips $n/4$ times

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} 1/2^i = n \times \frac{1}{1 - \frac{1}{2}} = 2n$$

- amortized cost is $O(n)/n = 1$

2. Accounting Method

- for different operations, we "charge" a specific amount $\hat{c}_i$ different than their actual costs c_i
    - can be less or more
- when amortized cost is *more than* the actual then
    - we store the excess credit into the object
    - credit is stored for future use when the amortized cost is *less than* the actual
- how do we assign amortized costs?
    - the total amortized cost *must* be an upper bound on the actual cost

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

- thus the total credit in the data structure is

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$$

Stack Operations

- actual costs
  - `push` is 1
  - `pop` is 1
  - `multipop` is $\min(k, s)$ where s is the length of S
- amortized cost
  - `push` is 2
  - `pop` is 0
  - `multipop` is 0
- analysis
  - each object in the stack has 1 "coin" of credit on it because it costs 1 to push and 1 gets saved
  - the total credit for a stack is going to be nonnegative as we can never pop more than what the stack has

3. Potential Method

- similar to the *accounting method*, however instead of storing credit, we store "potential"
- the potential is stored with the entire data structure instead of just a single object
- c_i is the actual cost
- D_i is data structure after the ith operation to D_{i-1}
- $\phi(D_i)$ is the potential associated with D_i
- $\hat{c}_i$ is the amortized cost of the ith iteration and is defined as $\hat{c}_i = c_i + \phi(D_i) + \phi(D_{i-1})$
- the total amortized cost is

$$\sum_{i=0}^{n} \hat{c}_i = \sum_{i=1}^{n} [c_i + \phi(D_i) - \phi(D_{i-1})] = \sum_{i}^{n} c_i + \phi(D_n) - \phi(D_0)$$

Stack Operations

Let the potential of a stack ϕ be the *number of elements* in the stack.

Actual cost		+	Potential diff	=	Amortized cost	
PUSH	1		+1		PUSH	2
POP	1		-1		POP	0
MULTIPOP	min(k, S)		-min(k,S)		MULTIPOP	0

Binary Counter

Let the potential of the counter ϕ be the *number of the 1's* in the counter.

Graphs

$$G = (V, E)$$

Algorithms

- finding cycles
- connected
- traversals: BFS, DFS
- topological sort
- strongly connected components

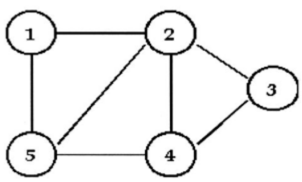
S'more Terminology

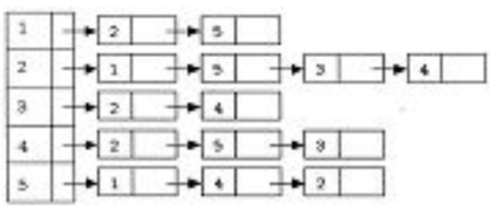
- in a directed graph, a path $< v_0, v_1, \ldots, v_k >$ forms a **cycle** if $v_0 = v_k$ and the path contains at least one edge
  - a **self-loop** is a cycle of 1
  - a directed graph with *no self-loops* is a **simple** directed graph
- in an undirected graph, a path $< v_0, v_1, \ldots, v_k >$ forms a **cycle** if $k \geq 3, v_0 = v_k$ and $v_1, v_2, \ldots v_k$ rare distinct
- **acyclic** graphs have no cycles
  - if an acyclic graph is connected, it is a **tree**
- **degree** if a vertex in undirected graph is number of edges incident to it
  - **out-degree** and **in-degree** of directed graph is edges leaving it and entering it
- the **length of a path** is the number of edges on it
- a graph is **connected** if every pair of vertices is reachable through a path
  - a directed graph is **strongly** connected if *both* vertices can reach each others
    - directed graph may have strongly connected *components*

Representation

Adjacency List

- every vertex has its own linked list containing its adjacent nodes





- the total memory required for an *undirected graph* is $O(|V| + 2 * |E|)$

- the total memory required for an *undirected graph* is $O(|V| + 2 * |E|)$
  - where we have to count every edge twice
- the total memory required for a *directed graph* is $O(|V| + |E|)$

Adjacency Matrix

- a $|V| \times |V|$ matrix where $A[i, j] = 1$ if an edge exists between i, j
  - if it is directed, $A[i, j]$ denotes an edge from i to j
- $|V|^2$ memory
- this is better for dense graphs
- undirect graph will be symmetric along the diagonal

Graph Traversal

Breadth First Search (BFS)

- uses a queue to visit the source's neighbors first before going to their neighbors
- if it's an undirected graph, all vertices will be visited if the graph is connected
- if it's a directed graph, all vertices will be visited if it is strongly connected

Code

```
BFS(G, s)
for each vertex u ∈ V[G] − {s}
      do color[u] ← WHITE
         d[u] ← ∞
         π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← Ø
ENQUEUE(Q, s)
while Q ≠ Ø
      do u ← DEQUEUE(Q)
         for each v ∈ Adj[u]
             do if color[v] = WHITE
                   then color[v] ← GRAY
                        d[v] ← d[u] + 1
                        π[v] ← u
                        ENQUEUE(Q, v)
         color[u] ← BLACK
```

Analysis

- $O(V + E)$
- $O(V)$ because every vertex is enqueued at most once
- $O(E)$ because every vertex is dequeued at most once
  - we examine (u, v) only when u is dequeued
  - therefore we examine every edge at most twice if undirected
  - at most once if directed
- BFS finds the shortest path to each reachable vertex in a graph from a given source
  - the procedure BFS builds a BFS tree

Depth First Search (BFS)

- uses a stack to explore as far down a branch as possible before backtracking to explore other branches

Code

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY            ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]        ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK           ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

Analysis

- $\Theta(V + E)$
- similar to BFS, however this is a tight Θ since it is guaranteed to examine every vertex and edge by restarting from disconnected components
- another interesting property of DFS is that the search can be used to classify the edges of the input graph

DFS Edge Classification

1. **tree edges** are edges in the depth-first forest G_π
   - edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v)
2. **back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree
   - self-loops (only in directed graphs) are considered to be back edges
3. **forward edges** are those nontree edges (u, v) connecting a vertex u to a descendant v is a depth-first tree
4. **cross edges** are all other edges
   - they can go between vertices in the same depth-first tree, as long as
   - one vertex is not an ancestor of the other, or
   - they can go between vertices in different depth-first trees

Why is this useful?

- a directed graph is acyclic if and only if a depth-first search yields no *back edges*
- in a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge

Topological Sort

- a DFS can be used to perform a topological sort of a directed acyclic graph (DAG)
- a topological sort of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, b) then u appears before b in the ordering
  - if the graph is cyclic then no linear ordering is possible
- a topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right

Code

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times
                  f[v] for each vertex v
as each vertex is finished, insert it onto
               the front of a linked list
return the linked list of vertices
```

Cycle Detection

G has a cycle if and only if DFS detects a back edge

Shortest Paths

- our input its
  - a directed graph $G = (V, E)$
  - a weight function $w : E \rightarrow \mathbb{R}$
- **weight of a path** $p =< v_0, v_1, \ldots, v_k >$ is the sum of its edge weights
- **shortest path** from u to v is any path p such that $w(p) = \delta(u, v)$

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v|\} & \text{if a path } u \rightsquigarrow v \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

Variants

- **single-source**: find shortest path from a given *source* vertex s to every vertex $v \in V$
- **single-destination**: find shortest path to a given destination
- **single-pair**: find shortest path u to v
  - there is no way known to solve that's better in the worst case than single-source
- **all-pairs**: find shortest path from u to v for all $u, v \in V$

"Gotchyas"

Negative-weight Edges

- they are okay so long as no negative-weight cycles are reachable form the source
  - if we have a negative-weight cycle, just keep going around it and we get $w(s, v) = -\infty$ for all v on the cycle
  - some algorithms work only if there are *no* negative-weight edges in the graph

Can a path contain a cycle?

- a path can't contain a negative cycle because you can always loop it again to decrease the path length
- a path can't contain a positive cycle because you can remove it to decrease the path length
- a path also can't a zero-weight cycle
- thus paths do not have cycles

Optimal substructure

- **lemma**: any sub-path of a shortest path is also a shortest path
- **proof**: using "cut and paste"
  - suppose p is a shortest path from u to v where $\delta(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$
  - suppose there is a shorter path p'_{xy} such that $w(p'_{xy}) < w(p_{xy})$
  - thus we can get a $\delta(p') = w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) < p$ which contradicts p being a shortest path

Single Source Algorithm: Bellman Ford

- can have negative weighted edges (but no cycles)

Variable Conventions

- $d[v]$ is a **shortest-path estimate** from the source s to some v
  - initially $d[v] = \infty$
  - always maintain $d[v] \geq \delta(s, v)$
- $\pi[v]$ is the predecessor of v on a shortest path from s
  - if there's no predecessor then $\pi[v] = \mathrm{NIL}$ (this is also our initialization)
  - π induces a **shortest-path tree**

Initialization

All the shortest-paths algorithms start with INIT-SINGLE-SOURCE.

INIT-SINGLE-SOURCE*(V, s)*

for each *v* ∈ *V*

  d[*v*]←∞

  π[*v*] ← NIL

d[*s*] ← 0

Relax

Can we improve the shortest-path estimate (best seen so far) for v by going through u and taking (u, v)?

RELAX*(u, v,w)*

if *d*[*u*] + *w(u, v)*<=*d*[*v*]

then *d*[*v*] ← *d*[*u*] + *w(u, v)*

  π[*v*]← *u*

Code

BELLMAN-FORD*(V, E,w, s)*

INIT-SINGLE-SOURCE*(V, s)*

for *i* ← 1 to |*V*|-1   **// for each vertex in V**

  for each edge *(u, v)* ∈ *E*   **// all edges, in any order**

    RELAX*(u,v,w)*

for each edge *(u, v)* ∈ *E*

  if *d*[*v*] > *d*[*u*] + *w(u, v)*

    then return FALSE

return TRUE

The first **for** loop relaxes all edges |*V*|-1 times.

O*(VE+E)=O(VE)*

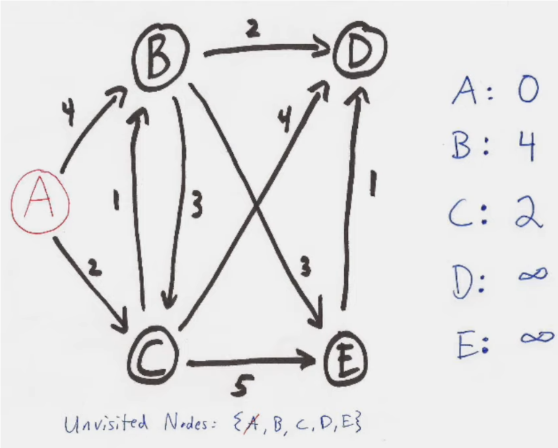
= O(V³)

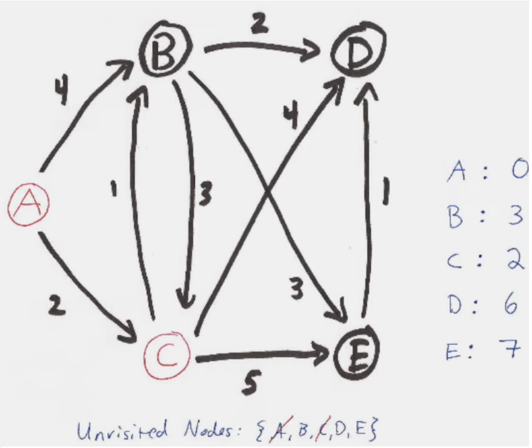
```
def bellman_ford(G, start):
```

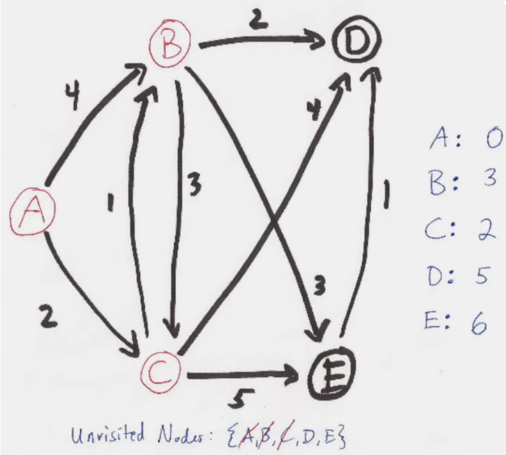
```
    shortest_paths = {}
    for node in G:
        shortest_paths[node] = infinity
    shortest_paths[start] = 0
    size = len(G)
    for _ in range(size - 1):
        for node in G:
            for edge in G[node]:
                cost = edge[0]
                to_node = edge[1]
                if shortest_paths[node] + cost < shortest_paths[to_node]:
                    shortest_paths[to_node] = shortest_paths[node] + cost
    # iterate once more and check for negative cycle
    for node in G:
        for edge in G[node]:
            cost = edge[0]
            to_node = edge[1]
            if shortest_paths[node] + cost < shortest_paths[to_node]:
                return 'INVALID - negative cycle detected'
    return shortest_paths
```

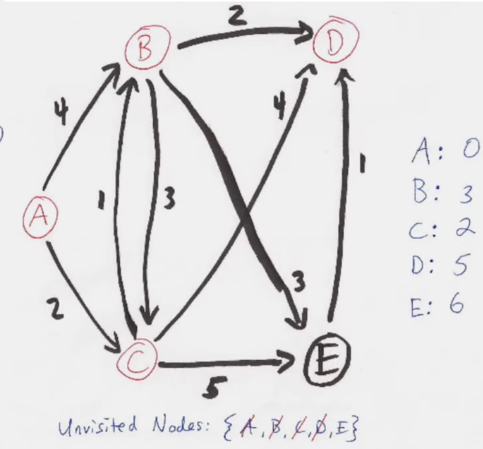
Single Source Algorithm: Dijkstra's Algorithm

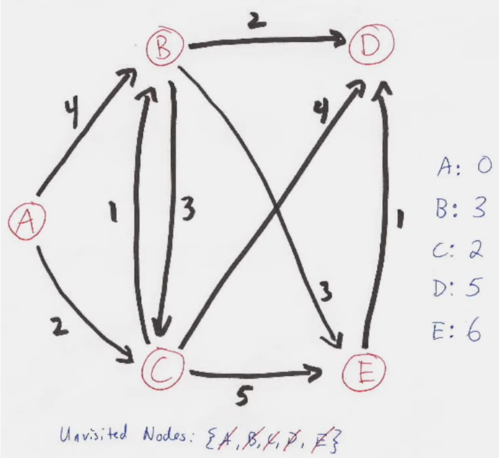
- no negative-weight edges
- is basically a weighted version of BFS
- instead of a FIFO queue, it used a priority queue using $d[v]$
- has 2 sets of vertices
    - S for vertices whose final shortest-path weights are determined
    - Q is a priority queue











- to pick the next vertex, pick the one that hasn't been chosen with the smallest $d[v]$
- if we implement the priority queue with a binary heap
  - $O(E \lg V)$
- proving greedy choice

Greedy Choice – pick the vertex with the smallest shortest path estimate (not including the vertices we are done with)
Assume we have a solution: we know the shortest path from s to every other vertex. “S” is the set of edges in the solution. If S does not contain the greedy choice at the last step, we can remove the non-greedy last edge added to S and add the greedy choice to S and get just as good a solution.

Understanding NP:

- **P:** A class of problems that can be solved by an algorithm in polynomial time.
- **NP:** Stands for "nondeterministic polynomial time." It includes problems for which a solution can be verified in polynomial time, even if finding the solution might take longer.

NP-Hard Problems:

A problem is NP-hard if solving it efficiently would also allow us to solve all NP problems efficiently. In other words, every NP problem can be reduced to an NP-hard problem in polynomial time.

Proving a Problem is NP:

1. **Show the Problem is in NP:**
    - To show a problem is in NP, demonstrate that any proposed solution can be verified in polynomial time.
    - For example, for the **Traveling Salesman Problem (TSP)**, given a route, it can be checked in polynomial time whether the route visits each city exactly once and returns to the starting city.
2. **Reduction to an NP-Complete Problem:**
    - A problem is **NP-complete** if it is both in NP and every problem in NP can be reduced to it in polynomial time.
    - To show a problem is NP-complete, show how an existing NP-complete problem can be reduced to it.
    - For example, reducing the **3-SAT** problem (which is NP-complete) to another problem can be used to demonstrate that the latter is also NP-complete.
3. **NP-Hard without being in NP:**
    - Some problems might be NP-hard but not in NP, especially if they can't be verified in polynomial time.
    - For example, the **Halting Problem** is NP-hard but not in NP.

Practical Examples:

1. **3-SAT:** A boolean satisfiability problem where you determine if there is an assignment that satisfies a boolean expression in conjunctive normal form with 3 literals per clause. It's an NP-complete problem.
2. **Subset Sum:** Given a set of integers and a target sum, determine if there is a subset of integers that sums to the target. It's NP-complete.

How to Prove NP-Completeness:

1. **Problem Verification:** Show that any proposed solution can be verified in polynomial time.
2. **Reduction:** Choose an existing NP-complete problem and show how it can be transformed into the new problem in polynomial time. This demonstrates that solving the new problem would also solve all NP problems.