Resum per l'examen final d'AP2 (Primavera 2021)

Algorithm Analysis

- Definicions de les diferents notacions-O:

Notació	Definició
$O(f(n))$	$\{g(n) : \exists k > 0, n_0 \in \mathbb{N} : g(n) \leq k \cdot f(n), \forall n \geq n_0\}$
$\Omega(f(n))$	$\{g(n) : \exists k > 0, n_0 \in \mathbb{N} : g(n) \geq k \cdot f(n), \forall n \geq n_0\}$
$\Theta(f(n))$	$\Omega(f(n)) \cap O(f(n))$

- Ordre d'infinits+noms:

Funció (decreasing order)	Nom comú
$n!$	Factorial
2^n	Exponencial
$n^d, d > 3$	Polinòmic
n^3	Cúbic
n^2	Quadràtic
$n\sqrt{n}$	----------
$n \log n$	Quasi-lineal // Linearítmic
n	Lineal
$\sqrt{n}$	Arrel d'n
$\log n$	Logarítmic
1	Constant

- Identitats útils:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \binom{n+1}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^{n} r^k = \frac{1 - r^{n+1}}{1 - r}$$

$$\log_b c = \log_b a \cdot \log_a c$$

- Producte de polinomis (Gauss):

$$PQ = P_L Q_L x^n + (P_R Q_L + P_L Q_R) x^{n/2} + P_R Q_R = P_L Q_L x^n + ((P_L + P_R)(Q_L + Q_R) - P_L Q_L - P_R Q_R) x^{n/2} + P_R Q_R$$

Master Theorem (Divide and Conquer)

Si partim el problema en a sub-problemes (*branching factor*) de mida $\frac{n}{b}$, i combinem els resultats d'aquest sub-problemes en $O(n^c)$, és a dir,

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^c),$$

llavors el temps $T(n)$ és:

$$T(n) = \begin{cases} O(n^c), & \text{si } c > \log_b a, \\ O(n^c \log n), & \text{si } c = \log_b a, \\ O(n^{\log_b a}), & \text{si } c < \log_b a. \end{cases}$$

Exemple: QuickSort. Fem una partició del vector sel·leccionant un pivot tal que uns quants elements siguin majors i la resta menors o iguals. La partició de Hoare ens ajuda bastant a fer això eficientment:

```
// Hoare's Partition
template <typename num>
int HoarePartition(vector<num> v, int left, int right) {
    // v[left..right]: segment to be sorted.
    // Output: The left part has elements <= than the pivot.
    // The right part has elements >= than the pivot.
    // Returns the index of the last element of the left part.
    num x = v[left];  // pivot
    int i = left - 1, j = right + 1;
    while (true) {
        do {
            i = i + 1;
        } while v[i] < x;
        do {
            j = j - 1;
        } while v[j] > x;
        if (i >= j) return j;
        swap(v[i], v[j]);
    }
}

template<typename num>
void QuickSort(vector<num> v, int left, int right) {
    // v[left..right]: segment to be sorted
    if (left < right) {
        int mid = HoarePartition<num>(v, left, right);
        QuickSort(v, left, mid);
        QuickSort(v, mid+1, right);
    }
}
```

L'algoritme QuickSort és de $O(n^2) \cap \Omega(n \log n)$, amb $T_{\text{avg}}(n) = O(n \log n)$. Alguns casos podrits poden empitjorar molt les coses: per exemple quan el pivot sempre és l'element més petit, worst case, o quan el vector està gairebé totalment ordenat o totalment ordenat.

Muster Theorem (Substract and Conquer)

Si partim el problema en a sub-problemes (*branching factor*) de mida $n-b$, i combinem els resultats d'aquest sub-problemes en $O(n^c)$, és a dir,

$$T(n) = a \cdot T(n-b) + O(n^c),$$

llavors el temps $T(n)$ és:

$$T(n) = \begin{cases} O(n^c), & \text{si } 0 < a < 1, \\ O(n^{c+1}), & \text{si } a = 1, \\ O(n^c a^{n/b}), & \text{si } a > 1. \end{cases}$$

Exemple: Torres de Hanoi. L'equació recursiva per el temps d'execució de l'algoritme de Hanoi és $T(n) = 2T(n-1) + O(1)$. Tenim, per tant, $a = 2$,$b = 1$, i $c = 0$, i per tant, $T(n) = O(n^0 2^{n/1}) = O(2^n)$.

ADT

- Binary Heap: es pot construïr un binary heap en temps lineal a partir d'un vector:

```
BinaryHeap(const vector<Elem>& items) {
    v.push_back(Elem()); // v is the vector holding the elements
    for (auto& e: items) v.push_back(e);
    for (int i = size()/2; i > 0; --i) bubble_down(i);
}
```

El podem fer servir per fer HeapSort:

```
template <typename T>
void HeapSort(vector<T>& v) {
    BinaryHeap<T> heap(v);
    for (T& e: v) e = heap.remove_min();
}
```

Grafs

- Estructures de dades per guardar:
  - Matriu d'adjacència: espai $O(|V|^2)$, per trobar un edge temps constant. Va bé per **grafs densos**.
  - Llista d'adjacència: espai $O(|E|)$, per trobar un edge (u, v) hem de buscar per tota la llista d'adjacència d'u. Va bé per **grafs escassos**.

Traversals i camins mínims

DFS

Algorisme d'exploració en profunditat, complexitat $O(|V| + |E|)$.

```
template<typename T>
using matrix = vector<vector<T>>;
using Node = int;
using graph = matrix<Node>;   // adjacency list

struct lifetime {
```

```
    int pre;
    int post;
    lifetime() {
        pre  = -1;
        post = -1;
    }
};

int visitTime = 0;
void previsit(Node u, vector<lifetime> &LT);  // is explained later
void postvisit(Node u, vector<lifetime> &LT); // is explained later

void explore(const graph &G, vector<bool> &visited, Node u, vector<lifetime>
&LT) {
    visited[u] = true;
    previsit(u, LT); // not needed
    for (Node v : G[u]) {
        if (not visited[v]) explore(G, visited, v, LT);
    }
    postvisit(u, LT); // not needed
}

vector<bool> reachable;// will contain all true after DFS
vector<lifetime> time; // will contain pre-visit and post-visit numbers

void DFS(const graph &G) { // O(|V|+|E|)
    int n = G.size(); // number of nodes
    reachable = vector<bool>(n, false);
    time = vector<lifetime>(n);
    for (Node u = 0; u < n; ++u) { // for each node in G
        if (not reachable[u]) explore(G, reachable, u, time);
    }
    // we could optionally print the order in which we traversed the graph (see
later)
}
```

Fent servir aquest algoritme, podem trobar, per exemple, les components connexes d'un graf:

```
void explore(const graph &G, Node u, int cc, vector<int> &ccnum) { // Different
signature than previous explore function
    // Input: G = (V,E) is a graph, cc is a Connected Component number
    // Output: ccnum[v] = cc for each vertex v in the same connected component as
u
    ccnum[u] = cc;
    for (Node v : G[u]) {
        if (ccnum[v] < 0) explore(G, v, cc, ccnum);
    }
}

vector<int> ConnComp(const graph &G, const vector<Node> &nodes) {
    // Input: G = (V, E) is a graph
    // Output: every vertex has a connected component number in the returned
vector
    int n = G.size(); // number of nodes
    vector<int> connectedComponents(n, -1);
    int cc = 0;
    for (Node u : nodes) { // for each node in G
```

```
        if (connectedComponents[u] < 0) {
            explore(G, u, cc, connectedComponents);
            ++cc;
        }
    }
    return connectedComponents;
}
```

Si volem recordar per on hem passat durant el DFS, només hem de fer servir les funcions de `previsit` i `postvisit`:

```
int visitTime = 0;

void previsit(Node u, vector<lifetime> &LT) {
    LT[u].pre = visitTime++;
}
void postvisit(Node u, vector<lifetime> &LT) {
    LT[u].post = visitTime++;
}
```

D'aquesta manera, podem construïr un arbre que passa per tots els vèrtexs del graf. Amb aquest algoritme també es pot descobrir si un graf té cicles. La funció `postvisit` també la podem fer servir per construïr una ordenació topològica d'un DAG: aquest plantejament també ordena els vèrtexs en funció del número de post-visita.

```
// ...
vector<Node> TSort;

void postvisit(Node u, vector<lifetime> &LT) { // different signature
    LT[u].post = visitTime++;
    TSort.push_back(u);
}
```

Una altra aplicació és trobar les **components fortament connexes** (CFC) d'un graf dirigit: aquests objectes tenen les següents propietats:

- Tot graf és un DAG de les seves components fortament connexes.
- Si la funció `explore` comença al vèrtex u, acabarà quan tots els vèrtexs que es poden visitar des d'u estiguin visitats.
- Si C i C' són dues components fortament connexes i hi ha un arc $C \to C'$ llavors el número de post-visita més gran de C és més gran que el número de post-visita més gran de C'.
- El vèrtex amb número de post-visita més alt és d'una CFC font.

L'algoritme que volem per trobar les CFC comença un DFS a partir d'un vèrtex en una CFC pou. Un vèrtex com aquest el podem trobar **revertint el graf**:

```
bool beforeNode(const Node &a, const Node &b) {
    return time[a].post > time[b].post;
}

graph reverse(const graph &G, const vector<Node> &nodes) {
    int n = G.size();
    graph GR(n);
    for (Node u : nodes) {
        for (Node v : G[u]) GR[v].push_back(u); // we reverse edge (u,v)
```

```
    }
    return GR;
}

void SCC(const graph &G, vector<int> &sccnum, vector<Node> &nodes) {
    // Input: G = (V, E) is a graph
    // Output: sccnum will contain the SCC numbers for all nodes u in G
    int n = G.size(); // number of nodes
    sccnum = vector<int>(n, -1);
    graph GR = reverse(G, nodes);
    DFS(GR); // computes post-visit number (and can sort in O(n) if postvisit
function creates a topological ordering)
    // If we want to sort the nodes ourselves with the postvisit numbers,
    sort(nodes.begin(), nodes.end(), beforeNode);
    sccnum = ConnComp(G, nodes);
    // If we want the nodes to be automatically sorted, we can directly use the
TSort vector
    // that the postvisit function leaves after execution of DFS
    sccnum = ConnComp(G, TSort);
}
```

Podríem fer el pas d'ordenar els nodes en funció de número de post-visita quan cridem la funció de post-visita, de fet, i aleshores es farà en temps lineal en $|V|$. De fet, amb cridar `ConnComp()` amb el vector `TSort` ja en tindríem prou.

L'algoritme DFS també es pot implementar sense ajuda de funcions externes, usant una pila (*stack*):

```
void DFS(const graph &G, Node source, vector<bool> &reachable) {
    // Input: G = (V, E) is a graph, source is a node in the graph
    // Output: reachable contains whether a node is reachable from u or not
    int n = G.size(); // number of nodes
    reachable = vector<bool>(n, false);
    stack<Node> S;
    S.push(source);
    while(not S.empty()) {
        Node u = S.top(); S.pop();
        if (not reachable[u]) {
            reachable[u] = true;
            for (Node v : G[u]) {
                if (not reachable[v]) S.push(v);
            }
        }
    }
}
```

BFS

Algoritme de cerca en amplada. Troba la distància mínima d'un vèrtexs a tots els que pot arribar. Té complexitat $O(|V| + |E|)$.

```
void BFS(const graph &G, Node orig, vector<int> &distances) {
    // Input: G = (V, E) is a graph, orig is the source block
    // Output: distances contains the shortest distance between orig and each
node
    int n = G.size(); // number of nodes
```

```
    distances = vector<int>(n, -1);
    distances[orig] = 0;

    queue<Node> q;
    q.push(orig);
    while (not q.empty()) {
        Node u = q.front();
        for (Node v : G[u]) {
            if (distances[v] < 0) {
                distances[v] = distances[u] + 1;
                q.push(v);
            }
        }
    }
}
```

Dijkstra

L'algoritme de Dijkstra troba el cost mínim i el camí de cost mínim des del vèrtex u fins a la resta de vèrtexs del graf, amb pesos no negatius als arcs.

```
using Weight = int;
int inf = INT_MAX;
using WN = pair<Weight, Node>;

void Dijkstra(const graph &G, const matrix<Weight> &W, Node orig, vector<int>
&distances, vector<Node> &ancestorInDAG) {
    // Input: G = (V, E) is a graph, W[u] is a list of weights from u to each of
its descendants
    // Output: distances contains the distance from orig to each node in G,
ancestorInDAG contains
    //         the ancestor of each node in a DAG formed by the minimum cost
paths from orig
    int n = G.size(); // number of nodes
    distances = vector<int>(n, inf);
    ancestorInDAG = vector<Node>(n, -1);

    distances[orig] = 0;
    priority_queue<WN, vector<WN>, greater<WN>> pq;
    pq.push({0, orig});

    while (not pq.empty()) {
        WN curr = pq.top(); pq.pop();
        Node u = curr.second;    // current node we are checking
        Weight w = curr.first;   // distances[u]

        for (int i = 0; i < (int)G[u].size(); ++i) {
            Weight h = W[u][i];  // not zero because u->G[u][:]
            Node v = G[u][i];
            if (w + h < distances[v]) {
                pq.push({w + h, v});
                distances[v] = w + h;
                ancestorInDAG[v] = u;
            }
        }
    }
}
```

L'algoritme té complexitat
$O((|V| + |E|) \log |V|) = O((n + m) \log n) = O(|E| \log |V|) = O(m \log n)$.

Bellman-Ford

Si hi hagués pesos negatius, però no cicles negatius, podem fer servir l'algorisme de Bellman-Ford. Té complexitat $O(|V| \cdot |E|)$ i fa el següent:

```
inf = 1e9;

void BellmanFord(matrix<int> &edges, Node orig, int n, vector<int> &distances,
vector<Node> &ancestorInDAG) {
    // Input: edges contains each edge in the graph, as in edge[i] = {origin,
destination, weight}
    //        n is the number of nodes, and orig is the node from where we want
to know the distances
    // Output: distances contains the minimum distance from orig to each node,
    //         ancestorInDAG contains the ancestor of each node in the tree
generated from the traversal
    distances = vector<Weight>(n, inf);
    ancestorInDAG = vector<Node>(n, -1);

    distances[orig] = 0;
    for (int i = 0; i < n - 1; ++i) {
        for (vector<int> &edge : edges) {
            Node u = edge[0];
            Node v = edge[1];
            Weight w = edge[2];
            if (distances[u] + w < distances[v]) {
                distances[v] = distances[u] + w;
                ancestorInDAG[v] = u;
            }
        }
    }

    // Next section is to check whether the graph has negative cycles
    for (vector<int> &edge: edges) {
        Node u = edge[0];
        Node v = edge[1];
        Weight w = edge[2];
        if (distances[u] + w < distances[v]) {
            cout << "there is a negative cycle containing nodes " << u << " and
" << v << endl;
            break;
        }
    }

    return;
}
```

Un cicle negatiu és un cicle en un graf tal que la suma dels pesos del cicle és negativa. Aquest podria produïr distàncies de $-\infty$ a l'algorisme, aplicant repetidament voltes al cicle. Per detectar aquests cicles, apliquem Bellman-Ford, i fem una ronda extra després del bucle principal comprovant que cap distància decreixi.

Camins mínims en DAGs

En qualsevol camí d'un DAG, els vèrtexs hi apareixen en ordenació topològica creixent. Per tant, amb una ronda de visita als arcs en ordre topològic en tindrem prou per trobar el camí mínim des d'un vèrtex fins un altre. És a dir, que la complexitat del següent algoritme serà $O(|V| + |E|)$:

```
inf = 1e9;

void DAGShortestPaths(const graph &DAG, const matrix<Weight> &W, Node orig,
vector<Weight> &dist, vector<Node> &ancestor) {
    // Input: DAG = (V, E) is a DAG (Directed Acyclic Graph), W is a weight
matrix, orig is the source node
    // Output: dist contains the minimum distance from orig to each node,
ancestor contains the ancestor in the tree of
    //         paths to other nodes from orig
    int n = DAG.size();
    dist = vector<Weight>(n, inf);
    dist[orig] = 0;

    // Now we linearize DAG from orig:
    TSort = vector<Node>(0);
    vector<bool> visited(n, false);
    vector<lifetime> times(n);
    explore(DAG, visited, orig, times);
    for (Node u : TSort) {
        for (Node v : DAG[u]) {
            Weight w = W[u][v];
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                ancestor[v] = u;
            }
        }
    }
}
```

En aquest cas, no importa si tenim o no arcs negatius. Si volem trobar els camins més llargs, canviem el signe de tots els pesos i trobem el camí més curt. Llavors, el cost del camí més llarg estarà canviat de signe, però els vèrtexs pels que passa estaran ben calculats. Com a alternativa, també podríem actualitzar les distàncies amb el màxim en comptes del mínim.

En resum, per **camins mínims** amb origen únic tenim els següents algoritmes:

Graf	Algoritme	Complexitat
Arcs no ponderats	BFS	$O(\|V\| + \|E\|)$
Arcs ponderats (costos no negatius)	Dijkstra	$O((\|V\| + \|E\|) \log \|V\|)$
Arcs ponderats (costos negatius i positius)	Bellman-Ford	$O(\|V\| \cdot \|E\|)$
DAGs	Ordenació Topològica (DFS amb postvisita)	$O(\|V\| + \|E\|)$

Per trobar els camins mínims entre totes les parelles d'un graf ponderat sense cicles negatius fem servir l'algoritme de Floyd-Warshall ($O(|V|^3)$), que fa servir programació dinàmica. Existeixen també altres algoritmes.

Arbres (grafs)

Un arbre és un graf connex (no dirigit) acíclic. Hi ha propietats que ens faciliten descobrir arbres:

- Qualsevol graf connex no dirigit té $m \geq n - 1$ arestes.
- Un graf no dirigit d'n vèrtexs és un arbre si i només si té $n - 1$ arestes.
- Un graf no dirigit és un arbre si i només si hi ha un únic camí entre dos vèrtexs qualssevol.

Arbres d'Expansió Mínims

Propietat del tall: siguin X un subconjunt d'arestes que formen part d'un MST. Llavors, si agafem qualsevol S subconjunt d'arestes tal que les arestes d'X no creuin entre S i $V \setminus S$, i agafem e l'aresta amb cost mínim que travessa d'S a $V \setminus S$, llavors $X \cup \{e\}$ és part d'un MST.

Aquesta propietat inspira un esquema per trobar MSTs:

```
X = {} // Set of edges of the MST
repeat |V|-1 times:
    pick a set S<V for which X does not cross between S and V-S
    let e<E be the minimum weight edge crossing between S and V-S
    X = X+{e}
```

Llavors, tenim dues estratègies principals:

Prim

Tenim un conjunt S de vèrtexs dins l'arbre, i afegim el vèrtex que surt d'S i va a $V \setminus S$ de cost mínim. Complexitat $O(|E| \log |V|)$.

```
struct NodeCost {
    Node node;
    Weight cost;
    NodeCost(Node n, Weight c) {
        node = n;
        cost = c;
    }
};

bool operator<(const NodeCost &a, const NodeCost &b) {
    return a.cost > b.cost;
}

void Prim(const graph &G, const matrix<Weight> &W, vector<Node> &ancestor,
Weight &total_weight) {
    // Input: G = (V, E) is a connected undirected graph, with edge weights W
    // Output: ancestor contains the ancestor of each node in a MST, except for
the root, which contains
    //         -1 as an ancestor, and total_weight contains the weight of the
MST
    int n = G.size();  // number of nodes
    vector<bool> visited(n, false);
    ancestor = vector<Node>(n, -1);
    total_weight = 0;
```

```
    priority_queue<NodeCost> pq;
    pq.push(NodeCost(0, 0));
    while (not pq.empty()) {
        NodeCost curr = pq.top();
        pq.pop();  // we take the node with minimum cost
        Node u = curr.node;
        Weight w = curr.cost;

        if (not visited[u]) {
            visited[u] = true;
            total_weight += w;  // we account for this edge's cost

            for (int i = 0; i < (int)G[u].size(); ++i) {  // we loop through all
u's neighbours
                Node v = G[u][i];
                Weight h = W[u][i];
                if (not visited[v]) {
                    ancestor[v] = u;
                    pq.push(NodeCost(v, h));  // we add u's neighbours to the
queue with their cost
                }
            }
        }
    }
}
```

Kruskal

Tenim un bosc F construït, i afegim al conjunt l'aresta de cost mínim entre dos dels arbres d'F. Complexitat $O(|E|\log|E|) = O(|E|\log|V|)$.

```
struct edge {
    Node origin;
    Node destination;
    Weight cost;
};

bool operator<(const edge &a, const edge &b) {
    return a.cost < b.cost;
}

Node findSet(Node u, vector<Node> &parent) {
    // Input: u is a node in the graph, parent contains the representative of
each node in the Kruskal forest
    // Output: returns the representative of u (its component root) and performs
a path compression meanwhile
    if (parent[u] != u) parent[u] = findSet(parent[u], parent);
    return parent[u];
}

void Kruskal(const vector<edge> &edges, int n, Weight &total_weight,
vector<edge> &X) {
    // Input: edges represents the edges of a connected, undirected graph
    //        n is the number of nodes
    // Output: total_weight contains the weight of the MST represented by the
edges in X
```

```
    sort(edges.begin(), edges.end()); // sort the graph edges in ascending cost
order
    vector<Node> parent(n);
    for (Node u = 0; u < n; ++u) parent[u] = u;
    total_weight = 0;
    for (edge &e : edges) {
        Node u = e.origin;
        Node v = e.destination;
        Weight w = e.cost;

        Node parent_u = findSet(u, parent); // representative of u
        Node parent_v = findSet(v, parent); // representative of v
        if (parent_u != parent_v) { // if the representatives are different,
they are in different connected components
            X.push_back({u, v, w}); // so we take the edge
            total_weight += w;
            parent[parent_u] = parent_v; // and we update the representatives
        }
    }
}
```

Idealment, aquest algoritme es fa usant una estructura de dades per guardar una col·lecció de conjunts disjunts. Té les operacions `makeset(x)` que crea el simplet $\{x\}$, `find(x)` que troba el representant del conjunt on es troba x, i `union(x,y)` que fa la unió dels conjunts que contenen els elements x i y. Kruskal faria $|V|$ crides a `makeset`, $2 \cdot |E|$ crides a `find` i $|V| - 1$ crides a `union`, de la següent manera:

```
function makeset(x): // O(1)
    parent(x) = x
    rank(x) = 0

function find(x): // O(log n)
    while (x != parent(x)) x = parent(x);
    return x

function union(x, y): // O(log n)
    r_x = find(x)
    r_y = find(y)
    if (r_x == r_y):
        return
    if (rank(r_x) > rank(r_y)):
        parent(r_y) = r_x
    else:
        parent(r_x) = r_y
        if (rank(r_x) == rank(r_y)):
            rank(r_y) += 1

function Kruskal(G, w):
    // Input: A connected undirected graph G = (V, E) with edge weights w_e
    // Output: An MST defined by the edges in X

    for all u<V: // O(|V|)
        makeset(u)

    X = {}
    sort edges in E by weight // O(|E|·log |E|) = O(|E|·log |V|)
```

```
    for all (u,v)<E, in ascending order of weight: //
2*|E|*O(find(x))=2*O(|E|log |V|)
        if (find(u) != find(v)):
            X = X+{(u,v)}
            union(u,v)
```

Qualsevol vèrtex *arrel* de rang k té almenys 2^k vèrtexs en el seu arbre. A més, si hi ha n vèrtexs en total, hi pot haver com a molt $n/2^k$ vèrtexs de rang k. Per tant, tots els arbres tenen alçada $\leq \log n$. Si puguéssim ordenar E en temps lineal, aleshores Kruskal seria $O(|E|)$, fent servir la funció `find` modificada per tenir cost amortitzat constant:

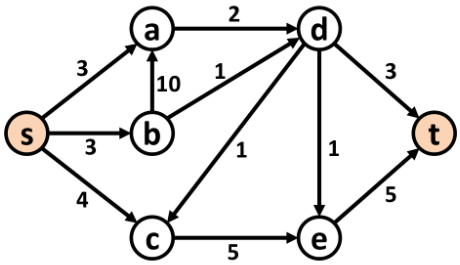
```
function find(x):
    if (x != parent(x)) parent(x) = find(parent(x))
    return parent(x)
```

MaxFlow-MinCut

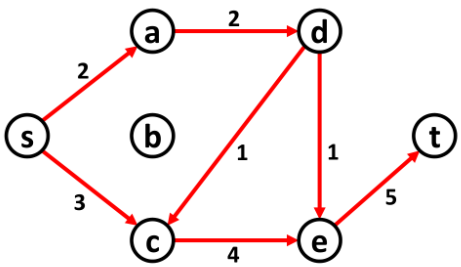
- **Flux màxim.** Volem determinar quin és el flux màxim que pot passar per una xarxa de canals (graf dirigit amb pesos=capacitat de cada canal).

El nostre objectiu és assignar un fluxe f_e a cada arc $e \in E$ tal que $0 \leq f_e \leq c_e$. Aquests fluxes hauran de satisfer que per tots els nodes (excepte per la font i el pou) que el flux que entra al noda és el mateix que el flux que en surt. La **mida d'un flux** és la quantitat total enviada des de la font s fins al pou, $mida(f) = \sum_{(s,u)\in E} f_{su}$. Suposem que tenim un flux sobre una xarxa. Si trobem un *augmenting path* (camí nou que afegeix més flux), llavors podem millorar el flux que teníem.

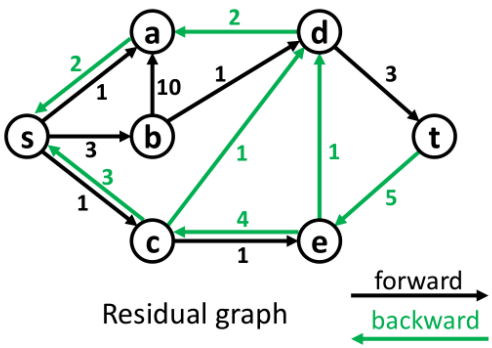
Treballarem amb el graf residual:



Graph



Flow



Residual graph

Buscarem camins de s fins a t. Per fer això farem servir l'algoritme de Ford-Fulkerson: anem buscant camins de flux màxim (els més llargs) fins que ja no en queden, i tindrem el flux màxim:

```
struct edge {
    Node origin;
    Node destination;
```

```
    Weight cap;
};

bool operator<(const edge &a, const edge &b) {
    return a.cap < b.cap;
}

void FordFulkerson(const vector<edge> &edges, Node source, Node sink,
matrix<Weight> &flow) {
    // Input: edges represents the edges of a graph G = (V, E)  together with the
edges' capacities
    //        source and sink are the source and target of the flow
    // Output: a flow that maximizes the size of a flow through G
    flow = matrix<Weight>(n, vector<Weight>(n, 0));
    for (edge &e : edges) {
        Node u = e.origin;
        Node v = e.destination;
        Weight c = e.cap;
        flow[u][v] = cap;
    }

    while (/*there is a path p from source to sink in the residual graph*/) {
        Weight width = /*minimum flow in p, aka the width of the path*/;
        for (edge e : p) {
            flow[e.origin][e.destination] -= width;
            flow[e.destination][e.origin] += width;
        }
    }
}
```

Trobar camins en el graf residual requereix $O(|E|)$ iteracions, si fem servir DFS o BFS. Per tant, com a molt farem $O(|V| \cdot |E|)$ iteracions.

- **Tall mínim.** Volem determinar quin és el nombre mínim de canals que hem de tapar per tal que s'interrompi el flux d'una font cap a un pou.

Un $(s,t)-$tall particiona els vèrtexs en dos conjunts disjunts, L i R, tals que $s \in L$ i $t \in R$. Per qualsevol flux f i qualsevol $(s,t)-$tall (L,R), tenim que

$$\text{size}(f) \leq \text{capacity}(L,R).$$

El següent teorema ens dóna una eina molt forta per trobar el flux màxim:

MaxFlow-MinCut Theorem. La mida del flux màxim és igual a la capacitat de l'$(s,t)-$tall més petit.

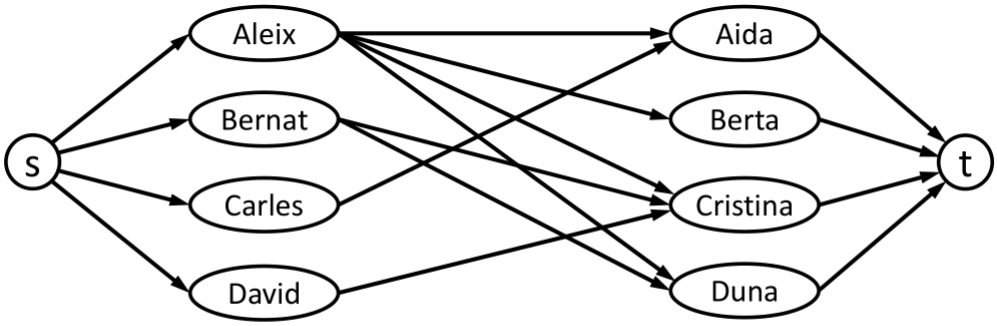
Augmenting Path Theorem. Un flux és màxim si i només si no admet cap augmenting path.

Com podem trobar per tant un tall amb capacitat mínima?

1. Solucionem MaxFlow amb Ford-Fulkerson.
2. Calculem L com el conjunt de vèrtexs als que es pot arribar des de s en el graf residual.
3. Definim $R = V \setminus L$.
4. El tall (L,R) és mínim.

Bipartite Matching

Hem de plantejar el problema com un problema de flux màxim:



Reduced to a max-flow problem with $c_e = 1$.

Podem garantir (per l'algoritme de Ford-Fulkerson) que els fluxos sempre seran enters.

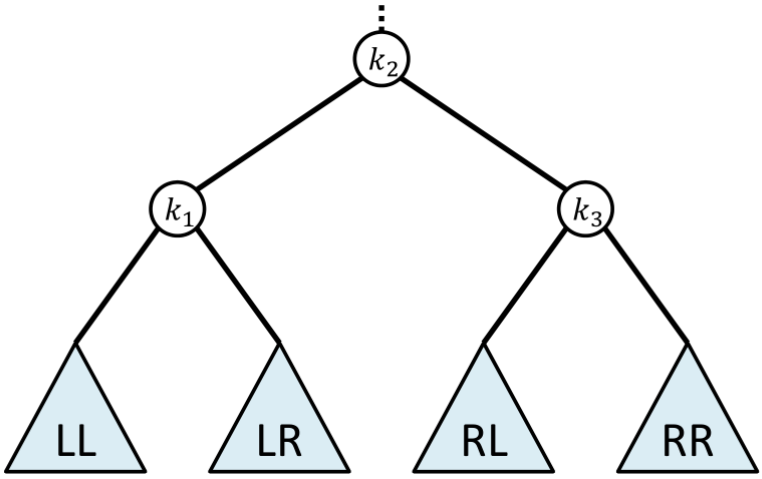
Arbres (ADT)

- **BST (Binary Search Tree).** Els elements del subarbre de l'esquerra són més petits que l'arrel, i els elements del subarbre de la dreta són més grans que l'arrel. Inserció simètrica (sempre manté balancejat) i logarítmica, però deleció logarítmica no simètrica (pot empitjorar molt l'alçada de l'arbre). L'accés també és logarítmic.
- **AVL Trees.** per millorar el fet que la deleció en un BST no és simètrica i afecta el balanceig de l'arbre, existeix aquest altre tipus d'arbre que manté l'alçada constant a $O(\log n)$. Un arbre AVL és un BST tal que per tot node, la diferència d'alçades entre el subarbre dret i el subarbre esquerre és com a molt 1.

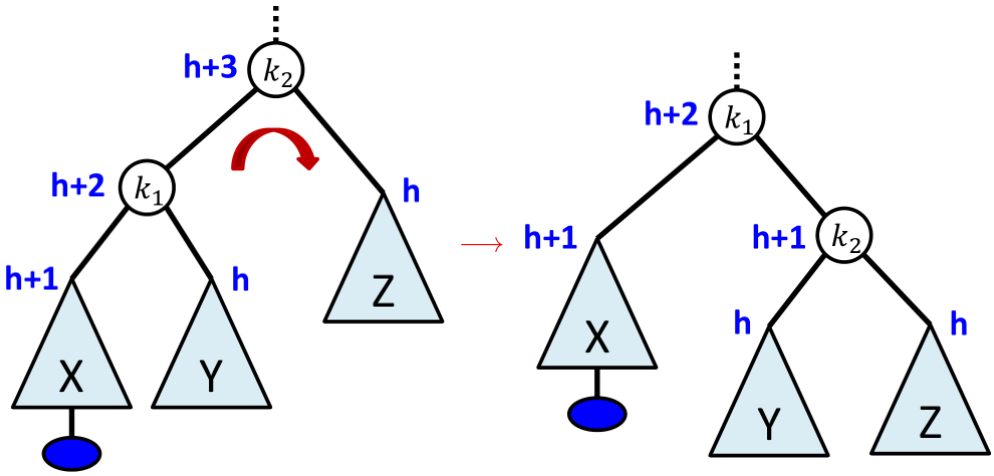
Teorema. Un arbre AVL de mida n té alçada $\Theta(\log n)$.

Teorema. L'alçada d'un arbre AVL amb n nodes interns satisfà que $h < 1.44 \log_2(n+2) - 1.328$.

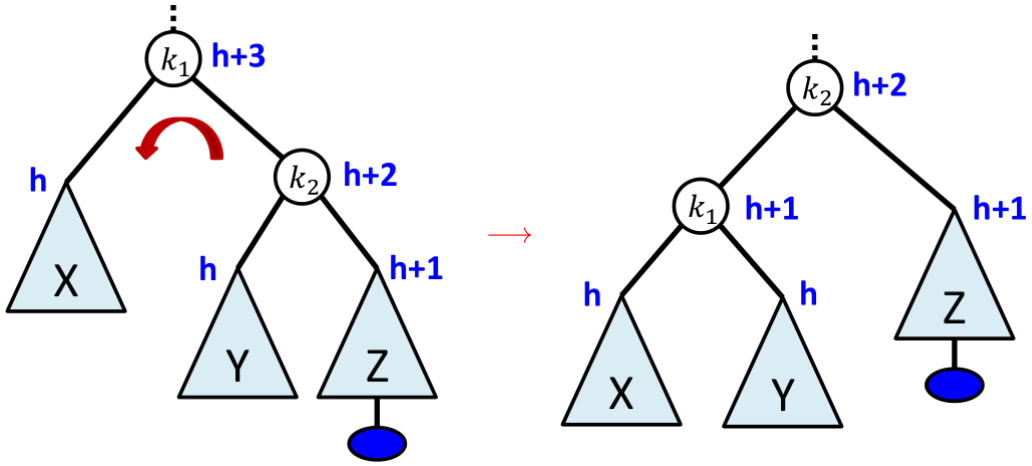
Una inserció en un arbre AVL pot caure en qualsevol d'aquests casos:



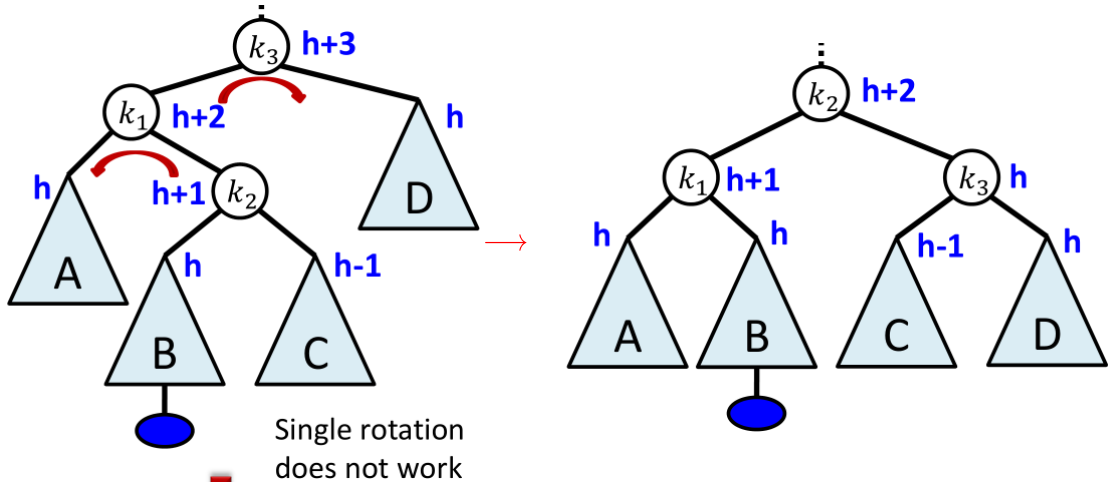
- **LL.**



- **RR.**

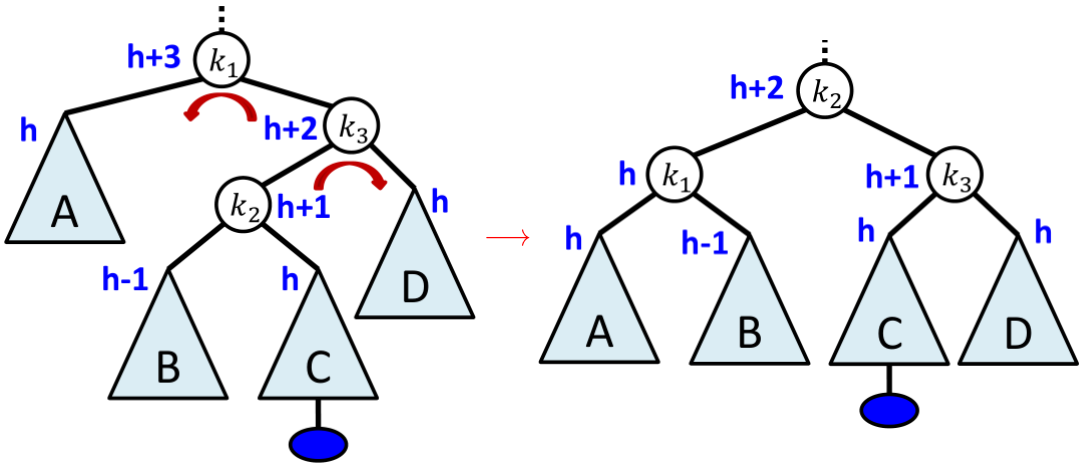


- **LR.**



Single rotation does not work

- **RL.**



Aquí em va fer mandra seguir traduïnt, sorry, així que a partir d'ara segueix en anglès

Hashing

A hash function h maps data of arbitrary size to a table of fixed size. We can calculate the position of an element x using the modulo operation, $h(x) \mod m$, where m is the size of the hash table. We should aim for random and uniform scattering of the elements in the hash table, and a consistent, deterministic function h. A usual hashing function for strings of size n is

$$h(x) = \sum_{k=0}^{n-1} x_k \cdot p^k,$$

where p is a prime number.

Collisions happen when for two different elements x_1 and x_2, we have that $h(x_1) \equiv h(x_2) \mod m$. To handle this situation, there are two main approaches:

1. The hash table contains lists as elements, so that when a collision happens, both elements go to the same "bucket" (buckets are usually linked lists or vectors).
    - In this approach, we define the **load factor** as $\lambda = \frac{\text{number of elements in hash table}}{\text{table size}}$. λ is the average length of a list, and a successful search takes on average $\lambda/2$ links to be traversed. Making the table size similar to the expected number of elements is a common strategy. When $\lambda > 1$, it is common to do **rehashing**.
    - **Rehashing.** When the table gets close to full, the probability of collision increases, and so does the cost of operations. Rehashing solves the situation by creating another hash table with a larger size and hashing all the elements again to put them in a cell in the new table (this obviously costs time of $O(n)$). The new size is usually close to $2n$ (ideally a prime number).
2. We use alternative cells in the hash table (double hashing, linear probing, ...).
    - If the cell is occupied, we look for an empty cell in the hash table and put the element there. Deletion then must be "lazy": deleted slots must be invalidated but not deleted. To avoid long trips looking for empty cells, the load factor should be under 0.5.
    - **Linear probing.** This strategy uses the next empty cell in the table to put the element in.
    - **Double hashing.** If there is a collision with hash function h, this strategy takes advantage of another hashing function g, and it searches for space in slots $h(x), h(x) + g(x), h(x) + 2g(x), \ldots$

Complexity analysis

Hash tables with buckets occupy space of $O(m+n)$, where m is the size of the table, and each slot has on average n/m elements. So, the average runtime to find an element is $O(n/m)$.

Cases	**Space:** $O(n+m)$	**Time:** $O(n/m)$
$m \gg n$	$O(m)$	$O(1)$
$n \gg m$	$O(n)$	$O(n)$
$m = O(n)$	$O(n)$	$O(1)$

The best strategy is to have $m = O(n)$ to keep search time constant while not wasting a lot of memory. To maintain $m = O(n)$ rehashing should be applied.

BST vs HashTable

Operation	**BST**	**HashTable**
Insertion/Deletion/Lookup	$O(\log n)$	$O(1)$
Sorted Iteration	In-order: $O(n)$	Needs an extra sorted vector: $O(n \log n)$
Hash Function	Not required	Required
Total Order	Required	Not required
Range Search	$O(\log n)$	$O(n)$

Cryptography

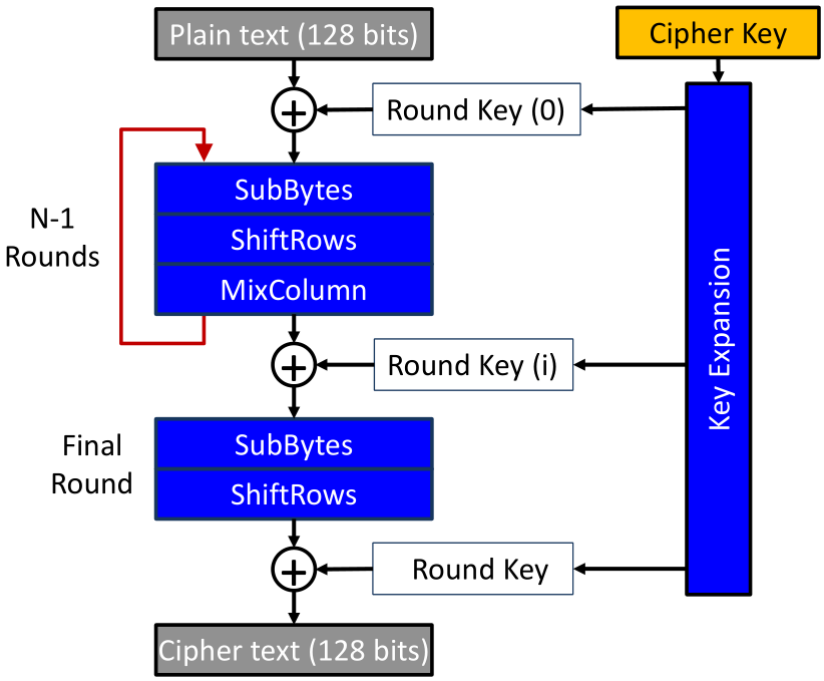
Secret key protocols

Symmetric encryption protocols. We see mainly two of them:

XOR encryption

The encryption and decryption functions are identical: the bitwise XOR function, $x \oplus r$, with a fixed secret key r. It is convenient that the bits in r are randomly generated. It is important to perform one-time padding, this is, changing the key every time we get a message, or the system will not be secure. If the key is short, the XOR will have to be applied many times to cover all of the message.

AES encryption



Public/Private key protocols

Each participat generates a private and a public key. Then, public keys are revealed to everyone. Each pair of (P_i, S_i) is a matched pair, this is, $M = S_i(P_i(M)) = P_i(S_i(M))$. Only the secret key holder can compute $S_i(X)$. If Bob wants to send Alice a message M, and Alice has the key pair (P_A, S_A), he can do so by calculating $X = P_A(M)$, and then Alice will decrypt the message by computing $M = S_A(X)$.

To create this kind of cryptosystem, we take advantage of modular arithmetic and Bézout's identity. We call the following system RSA.

- **Lemma.** If $d|a$ and $d|b$, and also $d = ax + by$ for integers x, y, then $d = \gcd(a, b)$.
- Extended Euclid's Algorithm:

```
function ExtendedEuclid(a, b):
    // Input: a and b are two positive integers
    // Output: it returns a triple (x, y, d) such that d = ax + by and d = gcd(a,
b)
    if (b == 0) return (1, 0, a)
    (x_aux, y_aux, d) = ExtendedEuclid(b, a mod b)
    return (y_aux, x_aux - floor(a/b)*y_aux, d)
```

- Let p, q be prime numbers, and $N = pq$. $\varphi(N) = (p-1)(q-1)$ is Euler's totient function of N, the number of integers less than N that are coprime with N. For any $e \in \mathbb{N}$ coprime with $\varphi(N)$,
  - The mapping $x \mapsto x^e$ is a bijection from $\{0, \dots, N-1\}$ to itself.
  - The inverse mapping can be obtained by finding d, the inverse modulo $\varphi(N)$ of e, and then, $(x^e)^d \equiv x \mod N$.

Then, the RSA cryptosystem works as following: Bob picks two large random primes p and q. Then, his public key is (N, e), where $N = pq$ and e is a small number coprime with $\varphi(N)$, and his private key is d, the inverse of e modulo $\varphi(N)$, computed using Euclid's extended algorithm. For Alice to send a message now, she has to compute $x^e \mod N$ and send the result y to Bob. Then, Bob can decrypt the message by computing $y^d \mod N$ (the message $x \in \{0, \dots, N-1\}$).

- One can mix symmetric and asymmetric cryptosystems to make communications more secure and faster.
- We could also use **Cryptographic Hash Functions**. A CHF maps arbitrary size inputs to a fixed size bit string. This is easy to compute, pre-image resistant and collision resistant. Popular CHFs include Message Digest (MD2, MD4, MD5, ..., 128-bit hash functions), Secure Hash (SHA-0, SHA-1, SHA-2, SHA-3) with hash values of 160 bits (SHA-1) or 256 bits (SHA-2), and some others.

Fast Fourier Transform

Polynomials can be represented by many different methods. Two of them are:

- Vector of coefficients: $P(x) = ax^2 + bx + c \mapsto (c, b, a)$
  - Addition is linear in the degree.
  - Multiplication is quadratic using brute force.
  - Evaluation is linear in the degree using Horner's method.
- Point-value representation:
  - FTA $\implies$ a polynomial of degree n, $A(x)$, is uniquely determined by its value at $n+1$ different points.
  - When a polynomial is represented as a collection of different point-value pairs,
  - Addition is linear in the degree.
  - Multiplication is linear in the degree.
  - Interpolation is quadratic using Lagrange's formula:

$$A(x) = \sum_{k=0}^{n-1} y_k \cdot \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)}$$

How to convert from coefficients to point-value representation? Given a polynomial of degree $n-1$, we perform Horner's rule on n different points. Then, the cost of this process is $O(n^2)$. But this can be done faster if we use a divide and conquer approach.

We can use pairs of $\pm x_i$ to do this process faster. $P(x)$ can be split into an even and an odd part, $P(x) = P_e(x^2) + x \cdot P_o(x^2)$, and then evaluating at x_i and at $-x_i$ are processes with many overlapping steps. Using this intuition, in order to be able to recurse (and really get $\pm$ pairs), we choose to evaluate the polynomial at the $n-$th complex roots of unity, being n the degree of the polynomial, and then we recurse. The algorithm is as follows:

```
function FFT(A, omega):
    // Input: A = (a_0,a_1,...,a_{n-1}), for n a power of 2
    //        omega is an n-th root of unity
    // Output: returns (A(1), A(omega), A(omega^2), ..., A(omega^{n-1}))
    if (omega == 1) return A

    // Even part of A
    (A_e(omega^0), A_e(omega^2), ..., A_e(omega^{n-2})) = FFT(A_e, omega^2)
    // Odd part of A
    (A_o(omega^0), A_o(omega^2), ..., A_o(omega^{n-2})) = FFT(A_o, omega^2)

    for (k = 0; k < n/2; ++k): // FFT Shuffling (some powers of omega are the
same)
        A(omega^k) = A_e(omega^{2k}) + omega^k*A_o(omega^{2k})
        A(omega^{k+n/2}) = A_e(omega^{2k}) - omega^k*A_o(omega^{2k})

    return (A(1), A(omega), ..., A(omega^{n-1}))
```

The runtime of the previous algorithm can be expressed as $T(n) = 2T(n/2) + O(n)$, which by the Master Theorem produces a time complexity of $T(n) = O(n \log n)$.

So, now we know that coefficient->point-value is a $O(n \log n)$ process. About interpolation, we have seen that the FFT algorithm computes

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix},$$

where $\omega = \exp(2\pi i/n)$. Let us call $F_n(\omega)$ this matrix. Thus, $y = F_n(\omega) \cdot a$. How about if we know y and want to obtain a (i.e., go from point-value to coefficients)? This is easy as a consequence of the following property of $F_n(\omega)$,

$$[F_n(\omega)]^{-1} = \frac{1}{n} \cdot F_n(\omega^{-1}),$$

and also the fact that if ω is an $n-$th root of unity, so is ω^{-1}. Then, to calculate coefficients from point-value representation, we just have to perform FFT of the values as if they were coefficients, using ω^{-1} instead of ω, and then divide the result by n. So, it is $O(n \log n)$ both ways.

In this way, we can achieve fast polynomial multiplication (not $O(n^2)$ anymore, but $O(n \log n)$) using the FFT algorithm: suppose we have polynomials A and B of respective degrees d_A and d_B. Let $d := d_A + d_B$. Then,

1. Selection Step:
    - Pick $\omega = \exp(2\pi i/n)$ such that $n > d$ is a power of 2.
2. Evaluation Step (FFT):
    - Compute $A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1})$.
    - Compute $B(1), B(\omega), B(\omega^2), \ldots, B(\omega^{n-1})$.
3. Multiplication Step (linear in the number of points):
    - Compute $C(\omega^k) = A(\omega^k) \cdot B(\omega^k)$ for $k = 0, \ldots, n-1$.
4. Interpolation Step (FFT):
    - Recover the coefficients of $C(x)$ using $\frac{1}{n} \cdot \mathrm{FFT}(\{C(\omega^k)\}_{k=0,\ldots,n-1}, \omega^{-1})$.