1. ANÀLISI D'ALGORISMES

$$1 < log(n) < \sqrt{n} < n < n\,log(n) < n^k < k^n < n!$$

$$Cas\ mig = \sum_{x \epsilon E}^{|x|=n} p(x) \cdot T(x) pitjor$$

$$Cas\ pitjor \rightarrow O(n) = Límit\ asimptòtic\ superior \rightarrow f \leq O(n)$$

$$\Theta(n) = Límit\ asimptòtic\ exacte \rightarrow f = \Theta(n)$$

$$Cas\ millor \rightarrow \Omega(n) = Límit\ asimptòtic\ inferior \rightarrow f \geq \Omega(n)$$

$$1 + 2 + 3 +... + n = \sum_{i=1}^{n} i = \frac{(n-1)n}{2}$$

$$2^0 + 2^1 + 2^2 +... + 2^n = \sum_{i=0}^{n-1} 2^i = 2^k - 1$$

Selection Sort

```
C/C++
// Retorno pos elem maxim part no ordenada (també funciona amb min)
int posicio_maxim (const vector<int>& v, int m) {
    int k = 0;
    for (int i = 1; i <= m; ++i)
        if (v[i] > v[k]) k = i;
    return k;
}
void ordena_seleccio (vector<int>& v, int n) {
    for (int i = n-1; i >= 0; --i) {
        int k = posicio_maxim(v, i);
        // Intercanvio element max pq estigui al final
        swap(v[k], v[i]);
}
```

Insertion Sort

```
C/C++
// Mantinc esquerra ordenada i vaig inserint els nombres dins
void ordena_insercio (vector<int>& v, int n) {
    for (int k = 1; k <= n-1; ++k) {
        int t = k-1;
        while (t >= 0 and v[t+1] < v[t]) {
            swap(v[t], v[t+1]);
            --t;
        }
    }
}
```

Teorema mestre recurrències

- **Subtractives:** $T(n) = a \cdot T(n - c) + g(n)$

$a < 1 \rightarrow \Theta(n^k)$

$a = 1 \rightarrow \Theta(n^{k+1})$

$a > 1 \rightarrow \Theta(a^{\frac{n}{c}})$

- **Subtractives:** $T(n) = a \cdot T(\frac{n}{b}) + f(n)$

$\alpha < k \rightarrow \Theta(n^k)$

$\alpha = k \rightarrow \Theta(n^k\, log(n))$

$\alpha > k \rightarrow \Theta(n^\alpha)$

2. DIVIDIR I VÈNCER

1. Subdividim el problema principal en casos més petits (però més senzills)
2. Resolem els subproblemes (cas base)
3. Combinem les resolucions dels subproblemes de forma intel·ligent.

Cerca binària

```
C/C++
int cerca_binaria(const vector<int>& a,int i,int j,int x) {
    // Divideixo la cerca en 2 per anar-me acostant
    if (i <= j) {
        int k = (i + j) / 2;
        if (x < a[k]) return cerca_binaria(a, i, k-1, x);
        else if (x > a[k]) return cerca_binaria(a, k+1, j, x);
        else return k;
    }
    else return -1;
}
```

MergeSort

```
C/C++
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    // Afegeixo de forma ordenada part esq i dret
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else              B[k++] = T[j++];
    }
    // Completo la part que quedi
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}

void mergesort (vector<elem>& T, int e, int d) {
    if (e<d) {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
    }
}
```

QuickSort

```
C/C++
// S'agafa primer com a pivot (però és podria agafar un aleatori o mediana)
int partition (vector<elem>& T, int e, int d) {
    // Agafo pivot i separo vector en 2, -pivot, +pivot
    elem x = T[e];
    int i = e;
    int j = d;
    while (true) {
        while (x < T[j]) --j;
        while (T[i] < x) ++i;
        if (i >= j) return j;
        swap(T[i], T[j]);
    }
}

void quicksort(vector<elem>& T, int e, int d) {
    if (e < d) {
    int q = partition(T, e, d);
    quicksort(T, e, q);
    quicksort(T, q + 1, d);
    }
}
```

Algorisme Karatsuba: Multiplica en cost $\Theta(n^{log_2 3})$

$x = [x_E][x_D] = 2^{n/2}x_E + x_D$ $y = [y_E][y_D] = 2^{n/2}y_E + y_D$

$$x \cdot y = 2^n x_E y_E + 2^{n/2}(x_E y_D + y_D x_E) + x_D y_D$$

Algorisme Strassen: Aplica Karatsuba en Matrius

Exponenciació

```
C/C++
// És més eficient fer x^2 * x^2 que x^4
double potencia (double x, int n) {
    if (n == 0) return 1;
    else {
        double y = potencia (x, n / 2);
        if (n % 2 == 0) return y * y;
        else return y * y * x;
    }
}
```

3. DICCIONARIS

- Asignar: añadir un elemento (llave, información) al diccionario. Si existía un elemento con la misma
llave, se sobrescribe la información.
- Eliminar: dada una llave, suprime el elemento que tiene dicha llave. Si no hay ningún elemento con
dicha llave, no hace nada.
- Presente: dada una llave, devuelve un booleano que indica si e diccionario contiene un elemento
con la llave dada.
- Búsqueda: dada una llave, devuelve una referencia al elemento con esa llave.
- Consulta: dada una llave, devuelve una referencia a la información de esa llave.
- Tamaño: devuelve el tamaño del diccionario.

Binary Search Tree (BST, Árboles binarios de búsqueda): Son estructuras arborescentes las cuales la
llave que contienen los nodos cumplen las siguientes propiedades:
- Los nodos mayores que la raíz van al subárbol derecho.
- Los nodos menores que la raíz van al subárbol izquierdo.

Consulta: supongamos que k es la llave que buscamos y x el la llave de la raíz:
- Si k = KEY(x) → Búsqueda completada.
- Si k < KEY(x) → Hacemos una llamada recursiva al subárbol izquierdo.
- Si k > KEY(x) → Hacemos una llamada recursiva al subárbol derecho.

Inserción:
- Si k = KEY(x) → Sobrescribimos la información.
- Si k < KEY(x) → Hacemos una llamada recursiva al subárbol izquierdo para insertarlo ahí.
- Si k > KEY(x) → Hacemos una llamada recursiva al subárbol derecho para insertarlo ahí.

Eliminación
- Si el nodo que queremos eliminar es una hoja (ambos subárboles están vacíos) → Eliminar el nodo.
- Si el nodo que queremos eliminar solo tiene un subárbol → sustituir el árbol existente por el padre.
- Si el nodo tiene los dos subárboles → Buscar el nodo más grande en el subárbol izquierdo y remplazarlo por el nodo que queremos eliminar.

4. PRIORITY QUEUE

Heap → es un árbol binario que:
- Todos los subárboles vacíos se encuentran en los últimos dos niveles del árbol.
- Si un nodo tiene un subárbol izquierdo vacío, el derecho también deberá estar vacío.

Hay dos tipos de heaps:
- Max-heaps → La prioridad de un elemento es mayor o igual que sus descendientes.
- Min-heaps → La prioridad de un elemento es menor o igual que sus descendientes.

5. GRAFS

DFS RECURSIU

```
C/C++
// Si no he visitat el node:
//     marco com a visitat
//     executo recursivament pels veins
void dfs_rec(const graph& G, int node, vector<boolean>& visitat,
             list<int>& llista) {
    if (!visitat[node]) {
        visitat[node] = true;
        llista.push_back(node);
        for (int node_adj : G[node]) {
            dfs_rec(G, node_adj, visitat, llista);
        }
    }
}

// Faig un DFS per cada node (per si no és conex)
// Si es vol fer només amb un node és pot fer directament
// dfs_rec(G, node, visitat, llista);
list<int> dfs_rec(const graph& G) {
    int n = G.size();
    list<int> llista;
    vector<boolean> visitat(n, false);
    for (int i = 0; i < n; ++i) {
        dfs_rec(G, i, visitat, llista);
    }
    return llista;
}
```

DFS ITERATIU

C/C++

```
// Per cada node afegeixo a la cua
// Mentre pila no buida
//     Miro top si no ha estat visitat
//         apunto llista
//         marco com a visitat
//         afegeixo a la cua els veins
list<int> dfs_iteratiu(const graph& G) {
    int n = G.size();
    list<int> llista;
    stack<int> pila_pendents;
    vector<bool> visitat(n, false);
    for (int i = 0; i < n; ++i) {
        // Si és vol fer només des de un node
        // es posa a la cua només l'inicial
        pila_pendents.push(i);
        while (!pila_pendents.empty()) {
            int node = pila_pendents.top();
            pila_pendents.pop();
            if (!visitat[node]) {
                visitat[node] = true;
                llista.push_back(node);
                for (int node_adj : G[node]) {
                    pila_pendents.push(node_adj);
                }
            }
        }
    }
    return llista;
}
```

BFS

```
C/C++
void bfs(const graph& G, int node_ini) {
    int n = G.size();
    queue<int> cua_pendents;
    vector<int> dist(n, -1);
    vector<int> node_procedent(n, -1);

    // Afegeixo node_ini a la cua
    cua_pendents.push(node_ini);
    distancies[node_ini] = 0;

    // Mentre cua no buida
    while (!cua_pendents.empty()) {
        int node = cua_pendents.front();
        // MIRAR SI ÉS CASELLA OBJECTIU!
        // Si ho és puc retornar, és la dist min!
        cua_pendents.pop();
        for (int node_adj : G[node]) {
            // Si node_adj no visitats
            if (dist[node_adj] != -1) {
                // Afegeixo cua
                // Li poso la dist a node_ini
                // Li poso el node del que ve
                cua_pendents.push(node_adj);
                dist[node_adj] = dist[node] + 1;
                node_procedent[node_adj] = node;
            }
        }
    }
}
```

DIJKSTRA

```
C/C++
void dijkstra(const graph& G, int node_ini) {
    int n = G.size();
    // priority_queue <PES, NODE> ordenat de forma creixent (menys pes
primer)
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> cua_pendents;
    vector<int> dist(n, numeric_limits<int>::max());
    vector<int> node_procedent(n, -1);

    // Afegeixo node_ini a la cua
    cua_pendents.push(0, node_ini);
    distancies[node_ini] = 0;

    // Mentre cua no buida
    while (!cua_pendents.empty()) {
        int node = cua_pendents.front();
        // MIRAR SI ÉS CASELLA OBJECTIU!
        // Si ho és puc retornar, és la dist min!
        cua_pendents.pop();
        for (int node_adj : G[node]) {
            // Si nova_dist node_adj < actual
            int new_dist = dist[node] + node_adj.second;
            if (new_dist < dist[node_adj]) {
                // Afegeixo cua
                // Li poso la new_dist al node
                // Li poso el node del que ve
                cua_pendents.push(new_dist, node_adj.first);
                dist[node_adj] = new_dist;
                node_procedent[node_adj] = node.first;
            }
        }
    }
}
```

TOPOLOGICAL SORT

```
C/C++
list<int> ordenacio_topologica(const graph& G) {
    int n = G.size();
    vector<int> grau_entrant(n, 0);
    stack<int> cua_pendents;
    list<int> ordre;

    // Inicialitzo els graus entrants
    for (int i = 0; i < n; ++i) {
        for (int node_adj : G[i]) {
            ++grau_entrant[node_adj];
        }
    }

    // Afegeixo els nodes sense graus entrants a la cua
    for (int i = 0; i < n; ++i) {
        if (grau_entrant[i] == 0) {
            cua_pendents.push(i);
        }
    }

    // Mentre cua no buida
    while (!cua_pendents.empty()) {
        int node = cua_pendents.top();
        cua_pendents.pop();
        ordre.push_back(node);

        for (int node_adj : G[node]) {
            // Redueixo el grau entrant dels nodes_adj
            // i si grau entrant és 0 l'afegeixo a la cua
            if (--grau_entrant[node_adj] == 0) {
                cua_pendents.push(node_adj);
            }
        }
    }
    return ordre;
}
```

PRIM | Minimum Spanning Tree

```
C/C++
int prim(const graph& G) {
    int n = G.size();
    vector<bool> visitat(n, false);
    priority_queue<P, vector<P>, greater<P>> cua_pendents;
    int n_visitats = 1;
    int cost_total = 0;

    // Afegeixo els nodes adjacents del node inicial a la cua
    visitat[0] = true;
    for (P node_adj : G[0]) {
        cua_pendents.push(node_adj);
    }

    // Mentre no hagi visitat tots els nodes
    while (n_visitats < n) {
        int node = cua_pendents.top().second;
        cua_pendents.pop();

        // Si no visitat
        if (!visitat[node]) {
            // Visitat
            // Augment cost_total i n_visitats
            visitat[node] = true;
            cost_total += cua_pendents.top().first;
            ++n_visitats;
            // Afageixo cua adjacents
            for (P node_adj : G[node]) {
                cua_pendents.push(node_adj);
            }
        }
    }

    return cost_total;
}
```

6. BACKTRACKING

- Són algoritmes que van construint la solució final a partir de solucions parcials (de forma recursiva)

- En el cas que la solució parcial no compleixi alguna restricció no continuem explorant aquella solució sinó que tirem enrere (BackTracking)

BRANCH & BOUND | Ramificacions i Poda

- És una versió millorada del Backtracking on quan fem una decisió fem una **ramificació** i avaluem aquelles decisions parcials.

- En el moment que preveiem que una branca inequívocament acabarà incomplint una restricció la **podem**, així evitem que continuï malgastant recursos.

- Tot i que és més eficient que el Backtracking respecte al temps d'execució, és més costosa en memòria.

7. P vs NP