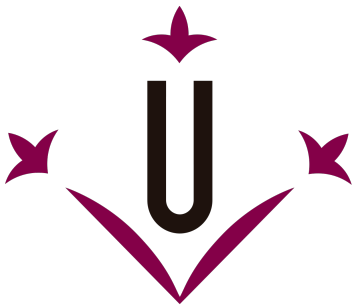

Universitat de Lleida

La illa de les hormones

Algoritmia i Complexitat

Escola Politècnica Superior

Grau en Enginyeria Informàtica

Yhislaine Nataly, Jaya Salazar –
Valeria Adriana, Escalera Flores –

1 de juny de 2025

Índex

1 Introducció

Aquesta pràctica té com a objectiu principal la implementació i anàlisi d'algorismes per a la resolució d'un problema de divisió d'equips en un entorn d'activitats lúdiques. En concret, es tracta d'organitzar un grup de $2n$ nois situats en posicions específiques a l'illa de les Hormones, dividint-los en dos equips, roig i blau, amb n participants cadascun. L'objectiu és aconseguir que la distància euclidiana entre el parell més proper de nois de diferents equips sigui la més gran possible, garantint així que els equips estiguin tan separats com sigui viable.

El problema s'ha de resoldre mitjançant l'elaboració d'un programa en Python (i opcionalment en Haskell) que llegeixi les dades d'un fitxer d'entrada i generi la solució en un fitxer de sortida. La solució ha d'incloure la distància màxima entre equips i un dels equips assignats. Cal analitzar tant el disseny recursiu com l'iteratiu dels algorismes principals, a més d'avaluar-ne el cost teòric i experimental.

La pràctica representa un 15% de la nota final i s'ha de presentar en parelles, incloent un informe detallat i el codi font, seguint les indicacions específiques de l'enunciat **sample**.

2 Anàlisi del problema

El problema consisteix a dividir $2n$ nois, situats en posicions concretes dins d'una illa, en dos equips (roig i blau) de n membres cadascun. L'objectiu és que la distància euclidiana entre el parell de nois més proper de diferents equips sigui la més gran possible. Això assegura que els equips estiguin el màxim de separats i, per tant, no s'interfereixin durant les activitats.

2.1 Exemple del PDF

Considerem l'exemple següent, extret de l'enunciat:

```
Entrada:
2
1 2
2 1
1 1
2 2
```

Aquí, $n = 2$, i tenim quatre nois amb les següents posicions:

- Noi 1: $(1, 2)$
- Noi 2: $(2, 1)$

- Noi 3: $(1, 1)$
- Noi 4: $(2, 2)$

Una possible divisió òptima és l'equip roig amb els nois 1 i 2, i l'equip blau amb els nois 3 i 4.

2.2 Càlcul de la solució

Per trobar la solució, cal calcular la distància euclidiana entre tots els possibles parells de nois que pertanyen a equips diferents. La distància euclidiana entre dos punts (x_1, y_1) i (x_2, y_2) es calcula com:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

En aquest cas, per la divisió proposada:

- Distància entre el noi 1 $(1, 2)$ i el noi 3 $(1, 1)$: $d = 1$
- Distància entre el noi 1 $(1, 2)$ i el noi 4 $(2, 2)$: $d = 1$
- Distància entre el noi 2 $(2, 1)$ i el noi 3 $(1, 1)$: $d = 1$
- Distància entre el noi 2 $(2, 1)$ i el noi 4 $(2, 2)$: $d = 1$

La distància mínima entre equips és 1, que és la màxima possible per a aquesta configuració. Aquesta és la solució que cal retornar.

2.3 Què és la distància euclidiana?

La distància euclidiana és la mesura de la distància "en línia recta"entre dos punts en un espai euclidià (com el pla o l'espai tridimensional). En dues dimensions, per als punts (x_1, y_1) i (x_2, y_2), la fórmula és:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Aquesta mesura és fonamental en aquest problema perquè determina la proximitat entre membres de diferents equips, i per tant, serveix per maximitzar la separació entre els equips.

2.4 Resum de l'estratègia

La solució òptima requereix provar totes les possibles divisions dels nois en dos equips iguals i, per cada divisió, calcular la distància mínima entre membres de diferents equips.

Finalment, es tria la divisió que maximitza aquesta distància mínima. Com que el nombre de combinacions és molt gran, cal buscar algorismes més eficients que la força bruta per resoldre el problema en casos grans.

3 Disseny dels algorismes

3.1 Backtracking

Aquest programa implementa un algorisme de **backtracking amb poda** per dividir $2n$ punts en dos equips de n jugadors de manera que la **distància mínima entre jugadors de diferents equips sigui la més gran possible**.[1]

El recorregut explora totes les formes d'assignar jugadors als dos equips, evitant combinacions no vàlides gràcies a tècniques de **poda**:

1. **Tamaño màxim**: si un equip ja té més de n jugadors, es deté.
2. **Impossibilitat futura**: si amb els jugadors restants no es poden completar els equips, es talla la branca.
3. **Cota inferior**: si la distància mínima actual entre equips és menor o igual a la millor trobada, es descarta la branca.

Pseudocodi

```
function backtrack(idx, equip1, equip2):
    if len(equip1) > n or len(equip2) > n:
        return

    if len(equip1) + (2n - idx) < n or len(equip2) + (2n - idx) <
    n:
        return

    if idx == 2n:
        min_dist = distancia m nima entre jugadors de equip1 i
   equip2
        si min_dist > millor:
            actualitzar millor
        return

    si equip1 i equip2 no buits:
        min_actual = distancia m nima actual
        si min_actual <= millor:
            return
```

```
18
19      // Provar assignar jugador idx a cada equip
20      afegir idx a equip1
21      backtrack(idx + 1, equip1, equip2)
22      treure idx de equip1
23
24      afegir idx a equip2
25      backtrack(idx + 1, equip1, equip2)
26      treure idx de equip2
```

Listing 1: Pseudocodi backtracking

Per evitar simetries, es fixa el **jugador 0 a l'equip 1** des del principi. (donde los equipos están simplemente intercambiados), se fija el **jugador 0 en el equipo 1** desde el principio. Esto reduce a la mitad el número de combinaciones a explorar.

Relació amb l'esquema general de backtracking

L'algorisme presentat segueix l'esquema general de **backtracking recursiu** explicat a classe:

```
1 def backtracking(done, todo, current):
2     if todo is empty:
3         return solution(done, current)
4     for elem in todo:
5         backtracking(done + elem, todo - elem, elem)
```

Listing 2: Esquema Backtracking

En el nostre cas:

- `done` es representa amb els equips `equip1` i `equip2`.
- `todo` es gestiona implícitament mitjançant l'índex `idx`.
- A cada pas, s'assigna el jugador actual a un dels dos equips.

Aquest enfocament és similar al de l'exercici 2 del document, on es divideixen actius entre dos socis. Tanmateix, mentre aquell problema busca igualar la suma dels valors, el nostre objectiu és **maximitzar la distància mínima** entre elements dels dos grups.

En ambdós casos s'apliquen tècniques de **poda** per reduir l'espai de cerca i evitar combinacions que no poden millorar la millor solució trobada.

3.2 Programació Dinàmica

El codi implementa un algorisme per dividir un conjunt parell de punts en dos equips iguals (n punts cadascun) de forma que la distància mínima entre qualsevol punt d'un

equip i qualsevol punt de l'altre equip sigui el màxim possible. És a dir, es vol **maximitzar la distància mínima** entre membres d'equips oposats.[2]

El codi segueix una aproximació clàssica de **programació dinàmica amb memòria (top-down)**:

- **Descomposició en subproblemes:** es defineix una funció recursiva `dp(i, mask1, mask2)` que intenta assignar el i-èssim punt a un dels dos equips.
- **Emmagatzematge de subsolucions:** s'utilitza un diccionari `memo` per guardar les solucions ja computades per a cada combinació d'estat $(i, mask1, mask2)$.
- **Reutilització de càlculs:** gràcies a la memòria, cada subproblema es resol només una vegada.
- **Cerca binària externa:** per maximitzar la distància mínima, es fa una cerca binària sobre la distància i es comprova si és possible assignar els equips mantenint la distància actual mínima.

Qualitat i bones pràctiques: El codi ha estat refactoritzat per seguir les bones pràctiques recomanades per l'eina `pylint`, obtenint una estructura modular, clara i documentada. Alguns punts destacats són:

- S'han definit **docstrings** a totes les funcions per documentar el seu propòsit.
- Es fa ús d'un **decorador** `count_calls` per comptar les crides a la funció `dist`, mostrant un ús idiomàtic del llenguatge Python.
- La generació de la matriu de distàncies es fa mitjançant un `generador`, aprofitant els recursos del llenguatge.
- L'entrada i sortida estan ben encapsulades en funcions específiques (`read_input` i `write_output`).
- S'evita l'ús de variables globals dins funcions internes (només es fa ús d'una global `min_dist` de forma controlada per la binària).
- El codi inclou tractament modular mitjançant la funció `main()` i comprovació amb `if __name__ == "__main__"`.

Funció `dp(i, mask1, mask2)`: retorna `True` si és possible assignar la resta de punts amb les màscares actuals, garantint que tots els punts assignats mantenen una distància mínima respecte l'altre equip.

```
def dp(i, mask1, mask2):
    key = (i, mask1, mask2)
    if key in memo:
        return memo[key]

    if bin(mask1).count('1') > n or bin(mask2).count('1') > n:
        return False
    if i == total:
        return bin(mask1).count('1') == n and bin(mask2).count('1
    ') == n

    # Intentar afegir el punt a l'equip 1
    if bin(mask1).count('1') < n:
        ok = True
        for j in range(total):
            if (mask2 >> j) & 1 and dist_matrix[i][j] < min_dist:
                ok = False
                break
        if ok and dp(i + 1, mask1 | (1 << i), mask2):
            memo[key] = True
            return True

    # Intentar afegir el punt a l'equip 2
    if bin(mask2).count('1') < n:
        ok = True
        for j in range(total):
            if (mask1 >> j) & 1 and dist_matrix[i][j] < min_dist:
                ok = False
                break
        if ok and dp(i + 1, mask1, mask2 | (1 << i)):
            memo[key] = True
            return True

    memo[key] = False
    return False
```

Listing 3: Esquema de la funció de programació dinàmica

```
left, right = 0.0, max(dist(p1, p2) for p1 in points for p2 in
   points)
for _ in range(40):
```

```
    mid = (left + right) / 2
    min_dist = mid
    memo.clear()
    if dp(1, 1, 0):
        left = mid
    else:
        right = mid
```

Listing 4: Cerca binària externa

Recomposició de la solució: Un cop trobada la distància màxima possible, la funció `reconstruct` reconstrueix quins punts pertanyen a cada equip, utilitzant la mateixa lògica de validació per distància i màscares.

Conclusió: Aquest enfocament top-down amb memòria resulta molt útil en casos on l'espai d'estats és gran i les transicions depenen de condicions complexes. L'ús eficient de Python i la bona estructura del codi fan que aquest exemple sigui no només correcte algorítmicament, sinó també **net, mantenible i ben valorat per eines com `pylint`**.

3.2.1 Ús de decoradors i generadors

El codi implementa tant un *decorador* com un *generador* per millorar l'estructura i l'eficiència del càlcul.

- **Decorador `compta_crides`**: Aquest decorador s'aplica a la funció `dist`, que calcula la distància euclidiana entre dos punts. La seva funció és comptar quantes vegades es crida aquesta funció al llarg de l'execució del programa. Això permet analitzar el cost computacional associat a aquest càlcul.

  ```
  @compta_crides
  def dist(p1, p2):
      ...
  ```

- **Generador `genera_matriu_distancies`**: Aquesta funció utilitza la paraula clau `yield` per generar progressivament les files de la matriu de distàncies entre punts. Això permet una construcció mandrosa de la matriu, la qual pot ser més eficient en termes de memòria per conjunts de dades grans.

```
def genera_matriu_distancies(punts):
    for i in range(total):
        yield [dist(punts[i], punts[j]) for j in range(total)
   ]
  \end{verbatim}

  La matriu es construeix amb:
  \begin{verbatim}
dist_matrix = list(genera_matriu_distancies(punts))
  \end{verbatim}
```

Finalment, el nombre total de crides a `dist` es mostra per pantalla amb:

```
print(f"Nombre de crides a dist: {dist.count}")
```

3.2.2 Haskell

Aquest codi Haskell resol el problema de dividir $2n$ punts en dos equips de n membres cadascun de manera que es maximitzi la **distància mínima** entre qualsevol parella de punts d'equips oposats.[1].

Trobar una partició dels punts en dos equips iguals tal que la distància mínima entre punts de diferents equips sigui la màxima possible. Això es fa mitjançant una combinació de **programació dinàmica amb memòria** i **cerca binària** sobre la distància mínima.

El cor de l'algorisme és la funció:

```
dp :: Int -> Mask -> Mask -> Int -> Int -> [[Double]] ->
   Double -> Memo -> (Bool, Memo)
```

Listing 5: Signatura principal de la funció de programació dinàmica

que intenta assignar cada punt al primer o al segon equip mentre comprova que totes les distàncies entre equips siguin majors o iguals a una distància mínima donada.

Els punts clau de programació dinàmica que es fan servir són:

- **Descomposició en subproblemes:** es considera, per a cada punt, totes les possibles assignacions als equips.
- **Emmagatzematge de subsolucions:** es guarda el resultat de cada estat en un `Map` (diccionari) per evitar recomputació.

[1] Per aquest codi es va crear un Checker apart anomenat `hsChecker.py`

- **Top-down amb memòria:** l'estructura no és iterativa sinó recursiva amb memòria (memoització).
- **Codificació eficient de l'estat:** es fa servir màscares de bits (`Int`) per representar els membres de cada equip.
- **Cerca binària:** s'utilitza per buscar la distància mínima màxima possible.

La funció `reconstruct` permet reconstruir la partició dels punts a partir de les decisions preses a `dp`.

El programa també utilitza un **functor** per aplicar la funció `fmap` sobre la llista d'índexs dels jugadors abans d'escriure'ls a l'arxiu de sortida, de manera que suma 1 a cada índex (ja que la numeració interna comença en 0, però l'entrada i sortida utilitzen numeració començant per 1). Aquest ús del functor permet aplicar una transformació de manera elegant i concisa sobre una estructura de dades (en aquest cas, una llista) sense haver d'escriure explícitament un bucle.

```
writeOutput :: FilePath -> (Double, [Int], [Int]) -> IO ()
writeOutput file (dist, team1, _) = do
    withFile file WriteMode $ \h -> do
        hPrintf h "%.6f\n" dist
        hPutStrLn h $ unwords (fmap show team1)  -- fmap =
  functor style
```

Listing 6: Ús del functor a `writeOutput`

Aquest codi implementa una solució eficient a un problema NP-difícil mitjançant:

- una cerca binària per optimitzar la distància mínima entre equips;
- programació dinàmica amb màscares de bits i memòria per tractar els subconjunts;
- una estructura top-down amb memoització per evitar càlculs repetits;
- ús de `fmap`, un exemple d'aplicació de functors, per manipular llistes de forma funcional i elegant.

Tot i no seguir exactament l'esquema iteratiu del PDF de la pràctica (amb matrius `dp[i][j]`), sí que utilitza una forma equivalent i habitual de programació dinàmica en problemes de partició de conjunts.

4 Anàlisi de costos

4.1 Anàlisi teòric

4.1.1 Backtracking

L'algorisme utilitza **backtracking amb poda** per explorar totes les particions possibles (evitant simetries fixant un jugador inicial a un equip). Per a cada assignació completa, es calcula la mínima distància entre un jugador d'un equip i un de l'altre, i es manté la millor trobada fins al moment.

Cost temporal en el pitjor cas:

- El nombre total de particions possibles és $\binom{2n}{n}/2$, però es redueix gràcies a la poda i a la fixació inicial.
- Per cada partició completa, es calcula una distància mínima entre n^2 parelles $\Rightarrow \mathcal{O}(n^2)$ per fulla.
- Cost total teòric en el pitjor cas (sense poda): $\mathcal{O}\left(\binom{2n}{n} \cdot n^2\right)$.

La poda redueix considerablement l'espai de cerca quan:

- Ja no hi ha prou jugadors per completar un equip.
- La distància mínima actual és inferior o igual a la millor trobada.

4.1.2 Dynamic Programming

L'algorisme combina cerca binària amb programació dinàmica sobre màscares de bits per assignar punts a dos equips.

- La cerca binària fa ~40 iteracions.
- La funció `dp(i, mask1, mask2)` pot explorar fins a $\mathcal{O}(n \cdot 2^{2n})$ estats.
- Cada estat requereix $\mathcal{O}(n)$ operacions per comprovar distàncies.
- El cost temporal total és $\mathcal{O}(40 \cdot n^2 \cdot 4^n)$.

4.1.3 Haskell

S'implementa un programació dinàmica i Binary Search. **Cost temporal en el pitjor cas:**

- Hi ha $\mathcal{O}(2^{2n})$ combinacions de mascaretes (una per a cada equip).

- Per cada estat, es fan fins a $2n$ comprovacions de distància.
- La cerca binària fa $k = 40$ iteracions per assolir precisió doble (coma flotant).
- **Cost total estimat:** $\mathcal{O}(k \cdot n \cdot 2^{2n})$.

Optimitzacions clau:

- Reducció de simetria: el jugador 0 sempre va al primer equip.
- Representació compacta d'equips amb enters (bitmask).
- Ús de memoització per evitar càlculs redundants.
- Cerca binària per evitar calcular totes les particions.

Cost computacional (temps i espai) dels algorismes.

4.2 Anàlisi experimental

4.2.1 Backtracking

S'ha observat empíricament que:

- Per valors petits de $n \leq 6$, el programa s'executa en menys d'un segon.
- A partir de $n = 7$, el temps de resposta augmenta significativament, degut a la combinatòria exponencial del problema.
- Amb la poda activa, el temps es redueix notablement comparat amb una cerca exhaustiva sense optimitzacions.

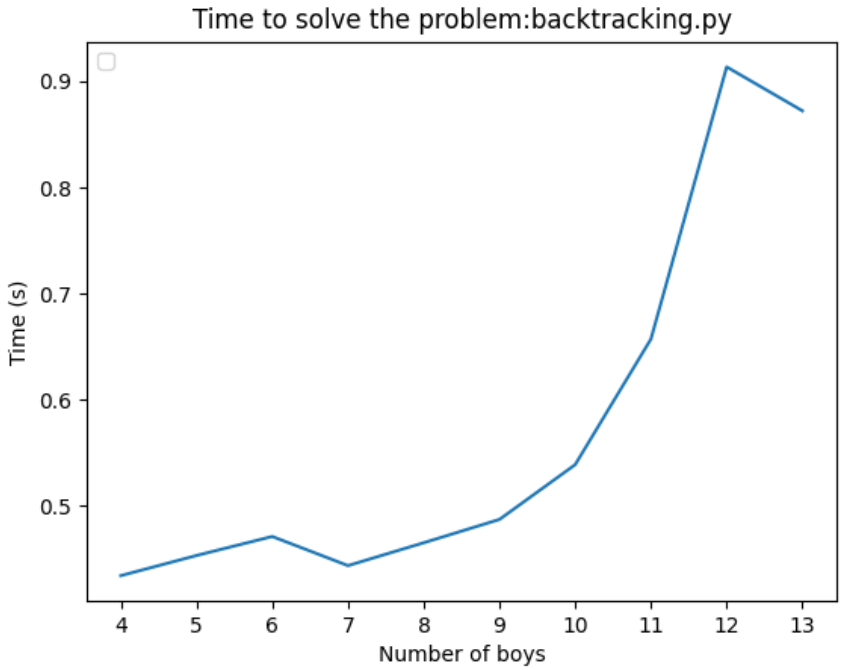

Figura 1: Gràfica Backtracking amb poda

4.2.2 Dynamic Programming

S'ha observat empíricment que:

- Per a valors petits del nombre de nois (de 4 a 7), el temps d'execució es manté gairebé constant, al voltant de 0.43 segons.
- A partir de 8 nois, el temps comença a augmentar de manera més notable, mostrant un creixement accelerat fins arribar a 1 segon amb 12 nois.

– Curiosament, en passar de 12 a 13 nois, el temps disminueix lleugerament, possiblement per variacions experimentals o característiques específiques de les dades d'entrada.

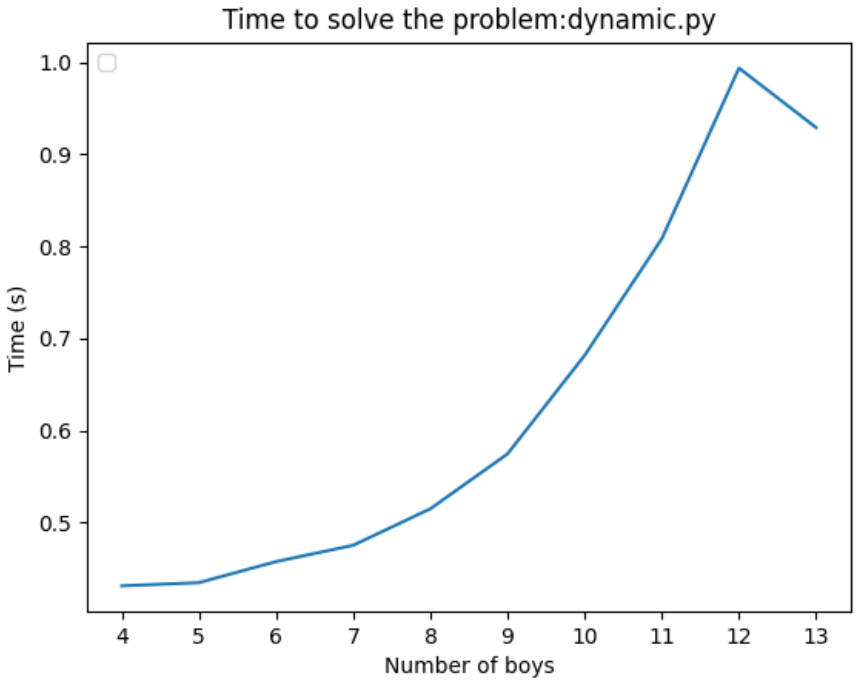

Figura 2: Gràfica Dyinamic Programming

4.2.3 Haskell

S'ha observat empíricament que:

– Per a valors petits del nombre de nois (de 4 a 7), el temps d'execució creix lentament, mantenint-se per sota dels 4 segons.

– A partir de 8 nois, el creixement del temps s'accentua, amb un augment progressivament més pronunciat.

– A partir de 10 nois, el creixement esdevé gairebé exponencial, arribant a superar els 12 segons amb 13 nois, la qual cosa indica una escalabilitat limitada del programa.

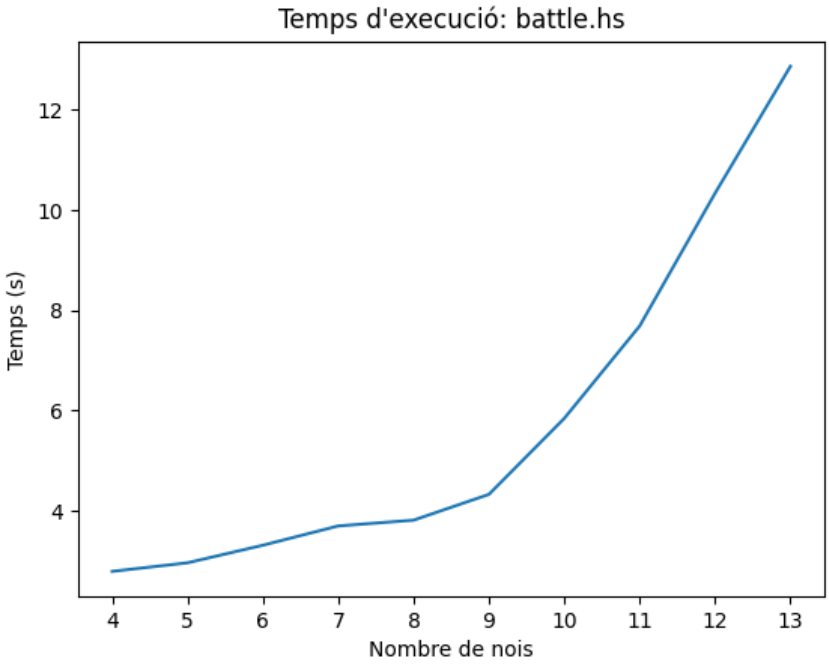

Figura 3: Gràfica del codi en Haskell

5 Conclusions

La resolució del problema de l'illa de les hormones ens ha permès aplicar diverses tècniques d'algorísmia avançada, com el backtracking amb poda i la programació dinàmica amb cerca binària. A través d'aquestes estratègies, hem pogut abordar de manera eficient un problema combinatori amb un espai de solucions exponencial. El backtracking ha estat útil per explorar l'espai de possibles divisions de forma exhaustiva però controlada, mentre que la programació dinàmica amb màscares i memoització ha ofert una alternativa més eficient, especialment per a instàncies de mida mitjana. Finalment, la implementació en Haskell ha demostrat com un llenguatge funcional pot ser adequat per expressar algoritmes complexos de forma concisa i elegant. Aquesta pràctica ha servit per aprofundir en el disseny i anàlisi d'algorismes, així com en l'optimització de rendiment i l'aplicació pràctica de conceptes teòrics.

Bibliografía

[1] J. Planes, *Algorithms and Complexity: May 2024 Exercises 4*, https://example.com/algorithms-exercises4.pdf, Documento leído en 2025. Fecha de creación decocónica, maig de 2024.

[2] J. Planes, *Algorithms and Complexity: May 2024 Exercises 6*, https://example.com/algorithms-exercises4.pdf, Documento leído en 2025. Fecha de creación decocónica, maig de 2024.