Resolución de Tarea 2 - Algoritmos Voraces (Fecha: 06 de Octubre de 2025)

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5651 - Diseño de Algoritmos I
Septiembre - Diciembre 2025
Estudiante: Junior Miguel Lara Torres (17-10303)

Tarea 2 (9 puntos)

Indice

Pregunta 1

El siguiente algoritmo propone 3 fases, cálculo del tiempo de finalización de cada canción, un ordenamiento de menor a mayor para este tiempo de finalización y una selección concreta de las canciones que forman parte de la solución final.

1.1 Consideraciones

- Cada canción se interpreta como un par $c_i = (t_i, d_i)$ donde t_i es el tiempo en el que empieza la canción en segundos y d_i la duracion de la cancion en

segundos.

- A nivel de segmentos de códigos se trabajará en C++ teniendo en cuenta la siguiente estructura de datos

```
struct Music {
    int id;     // Identificador
    int t;      // Inicio en segundos
    int d;      // Duración en segundos
    int finish; // Tiempo de finalización
};
```

1.2 Fase 1: Cálculo del tiempo de finalización

Para cada canción, se calcula el tiempo de finalización: $finish = t + d$. Esto porque el tiempo de finalización es esencial para decidir si una canción puede escucharse completa antes de que empiece la siguiente.

```
for (auto &song : songs) {
  song.finish = song.t + song.d;
};
```

1.3 Fase 2: Ordenamiento por tiempo de finalización

Se ordenan todas las canciones por su tiempo de finalización (finish) de menor a mayor, ya que ordenar por tiempo de finalización es la clave del algoritmo de selección de actividades, así se maximiza el número de canciones no solapadas que se pueden escuchar completas.

```
sort(songs.begin(), songs.end(), [](const Music &a, const Music &b) {
  return a.finish <= b.finish;
});
```

1.4 Fase 3: Selección de canciones

- Se selecciona la primera canción (la que termina más temprano).
- Para cada canción siguiente, si su inicio es mayor o igual a la finalización de la última seleccionada, se añade al resultado.
- Se actualiza el tiempo de finalización con la nueva canción seleccionada.

```
result.push_back(songs[0]);
int lastFinish = songs[0].finish;
for (int i = 1; i < songs.size(); ++i) {
  if (lastFinish <= songs[i].t) {
    result.push_back(songs[i]);
    lastFinish = songs[i].finish;
  }
};
```

1.5 Complejidad

1.5.1 Tiempo

- Para cada canción, se calcula $finish = t + d$. Esto requiere recorrer todas las canciones una vez: $O(n)$.
- Se utiliza *sort* para ordenar las canciones por su tiempo de finalización. El algoritmo de sort estándar de C++ (generalmente introsort) tiene complejidad $O(nlogn)$.
- Se recorre la lista ordenada una sola vez para seleccionar las canciones compatibles: $O(n)$.

Asi, tenemos complejidad $O(n) + O(n \cdot log(n)) + (n) = O(n \cdot log(n))$ para el tiempo.

1.5.2 Memoría

La memoria adicional es $O(n)$, ya que solo se almacenan las canciones y el resultado.

1.6 Programa

Para tener un programa 100% funcional con una dinámica de prueba por favor revisar el siguiente archivo music.cpp y un ejemplo de archivo de entrada songs.txt.

```cpp
struct Music {
  int id;     // Identificador
  int t;      // Inicio en segundos
  int d;      // Duración en segundos
  int finish; // Tiempo de finalización
};

vector<Music> schedulingMusic(vector<Music> &songs) {
  vector<Music> result;
  if (songs.empty())
    return result;

  for (auto &song : songs) {
    song.finish = song.t + song.d;
  }

  sort(songs.begin(), songs.end(), [](const Music &a, const Music &b) {
    return a.finish <= b.finish;
  });

  result.push_back(songs[0]);
  int lastFinish = songs[0].finish;
```

```
  for (int i = 1; i < songs.size(); ++i) {
    if (lastFinish <= songs[i].t) {
      result.push_back(songs[i]);
      lastFinish = songs[i].finish;
    }
  }
  return result;
}
```

Pregunta 2

Se tienen las definiciones del problema:

- T es un conjunto de tipos.
- F es el conjunto de todas las firmas de funciones posibles entre los tipos de T.
- $\mathcal{I}$ es la familia de subconjuntos **definitivos** de F.
- Un subconjunto $F' \subseteq F$ es definitivo si cada tipo de T que aparece en F' lo hace a lo sumo una vez como imagen (es decir, como el tipo de retorno de una función).

2.1 Demostración de que M_T es una Matroide

Para demostrar que $M_T = (F, \mathcal{I})$ es una matroide, debemos verificar que cumple con las tres propiedades fundamentales de las matroides.

- F es un conjunto finito.
- Cumple la propiedad hereditaria.
- Cumple la propiedad de intercambio.

2.1.1 F es finito

Al F depender del conjunto T de tipos, de tal forma que se generan firmas tomando a dos elementos arbitrarios de T, se tiene que $|F| = n^2$, donde $n = |T|$, por lo tanto, siempre y cuando T sea finito, tendremos F finito.

2.1.2 Propiedad Hereditaria

Se debe probar si un conjunto de firmas B es definitivo (independiente), entonces cualquier subconjunto $A \subseteq B$ también debe ser definitivo.

Tenemos que un conjunto es **no** definitivo si al menos un tipo se usa como imagen más de una vez. Pensemos en un conjunto B que sí es definitivo. Esto significa que no hay "conflictos de imagen" en B, entonces cada tipo aparece como imagen como máximo una vez. Si ahora tomamos un subconjunto A de B, estamos **eliminando** firmas. Al eliminar firmas, no podemos crear nuevos

conflictos. Si antes no había un tipo repetido como imagen, al quitar elementos, seguirá sin haberlo. Por lo tanto, si B es definitivo, cualquier subconjunto A de B también lo será.

2.1.3 Propiedad de Intercambio

Si tenemos dos conjuntos definitivos, A y B, y B tiene más firmas que A (es decir, $|B| > |A|$), entonces debe existir al menos una firma en B que no esté en A (llamémosla f) que podamos agregar a A para formar un nuevo conjunto $A \cup \{f\}$ que también sea definitivo.

- Pensemos en los tipos de imagen que usan los conjuntos A y B.
- Sea $Img(S)$ el conjunto de tipos de imagen (tipos de retorno) utilizados por un conjunto de firmas S.
- Como A es definitivo, todas las firmas en A tienen imágenes distintas. Por lo tanto, $|Img(A)| = |A|$.
- Del mismo modo, como B es definitivo, $|Img(B)| = |B|$.
- Dado que $|B| > |A|$, se deduce que $|Img(B)| > |Img(A)|$. Esto significa que el conjunto B debe estar usando al menos un tipo de imagen que el conjunto A no está usando.
- Por lo tanto, debe existir al menos una firma en B, digamos $f = (X \to Y)$, tal que su tipo de imagen Y no está en $Img(A)$. Como f tampoco está en A (si lo estuviera, Y estaría en $Img(A)$), podemos tomar esta firma f y agregarla a A.
- El nuevo conjunto $A \cup \{f\}$ seguirá siendo definitivo, porque la nueva firma f introduce un tipo de imagen Y que no estaba siendo utilizado por A. No se crea ningún conflicto.

Dado que $M_T = (F, \mathcal{I})$ cumple estas tres propiedades, podemos concluir que **es una matroide**.

2.2 Función de Costo para Maximizar el Potencial

Queremos encontrar un subconjunto definitivo que tenga el **potencial máximo**. El problema nos dice que el potencial de un conjunto de firmas es la suma de los potenciales individuales de sus firmas, y el potencial de una firma $X \to Y$ es $|Y|^{|X|}$.

Nuestra meta es maximizar el potencial total. El algoritmo voraz ordenará los elementos por peso **descendente** y elegirá en cada paso la firma de mayor "valor" que pueda agregar al conjunto sin que deje de ser definitivo. Por lo tanto, definir el peso (o costo) de una firma como su potencial.

2.2.1 Función de costo propuesta

Para una firma de función $f = (X \to Y)$, definimos su peso $w(f)$ como su potencial:

$$w(f) = |Y|^{|X|}$$

2.2.2 Algoritmo Voraz

Con esta función de peso, el algoritmo para encontrar el subconjunto definitivo de máximo potencial sería:

- Calcular el potencial (peso) para cada una de las $|F|$ firmas de funciones posibles.
- Ordenar todas las firmas en F en orden **descendente** según su peso.
- Inicializar un conjunto de solución $S = \emptyset$.
- Recorrer las firmas ordenadas. Para cada firma f:
  - Si agregar f a S mantiene el conjunto como definitivo (es decir, $S \cup \{f\} \in \mathcal{I}$), entonces agregar f a S.
  - Si no, descartar f.
- El conjunto S final es la solución.