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

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.

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

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

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;
};
```

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;
});
```

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;
  }
};
```

Complejidad

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.

Memoría

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

Programa

```
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;

  // Calcular tiempo de finalización
  for (auto &song : songs) {
    song.finish = song.t + song.d;
  }

  // Ordenar por tiempo de finalización
  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;
}
```