Notas de la clase 2 – técnicas algorítmicas

Francisco Soulignac

18 de marzo de 2019

Aclaración: este es un punteo de la clase para la materia AED3. Se distribuye como ayuda memoria de lo visto en clase y, en cierto sentido, es un reemplazo de las diapositivas que se distribuyen en otros cuatrimestres. Sin embargo, no son material de estudio y no suplanta ni las clases ni los libros. Peor aún, puede contener "herrorez" y podría faltar algún tema o discusión importante que haya surgido en clase. Finalmente, estas notas fueron escritas en un corto período de tiempo. En resumen: **estas notas no son para estudiar sino para saber qué hay que estudiar**.

Tiempo total: 230 minutos

Nota: no se incluye D&C ya que se ve en profundidad en AED2.

1. Backtracking (110 mins)

1.1. Problema genérico

- El input I define un conjunto $\mathcal{S}$ tal que todo elemento de $\mathcal{S}$ se puede describir como una cadena $S = s_0, \ldots, s_n$ en algún lenguaje L sobre un alfabeto Σ.
- Problema: encontrar (o determinar si existe) un elemento de $\mathcal{S}$ que satisfaga alguna propiedad.
- Búsqueda exhaustiva: recorrer todo $S \in \mathcal{S}$ hasta encontrar el que satisfaga la propiedad.
- Complejidad de búsqueda exhaustiva: $\Omega(|\mathcal{S}|)$. Esto es infactible si $|\mathcal{S}|$ es exponencial en $|I|$.

1.2. Ejemplos de problemas

- 2-PARTITION: dado un conjunto $X = \{x_1, \ldots, x_n\}$ de naturales, determinar si X puede partirse en conjuntos X_1, X_2 disjuntos tales que $\sum X_1 = \sum X_2$.
  - ▷ El conjunto X input define el conjunto $\mathcal{S}$ de partes de X.
  - ▷ Cada elemento de $\mathcal{S}$ se puede describir como una cadena binaria $S = s_1, \ldots, s_n$ tal que $x_i \in X$ si y sólo si $s_i = 1$.
  - ▷ El problema consiste en determinar si algún $S \in \mathcal{S}$ codifica un conjunto que sume $\sum X/2$.
  - ▷ Búsqueda exhaustiva: generar todos los subconjuntos de X y evaluar la suma. Complejidad $T(n) = \Theta(n2^n)$.[1]
- Fiesta (INDEPENDENT SET): el input es un conjunto $V = \{v_1, \ldots, v_n\}$ de posibles invitados a una fiesta, junto con una lista E de pares de invitados v_iv_j que se llevan mal. El problema es encontrar la lista de invitados más grande donde ningún par de invitados se lleven mal.
  - ▷ El conjunto V define el conjunto $\mathcal{S}$ de partes de V. Como antes, se puede codificar cada conjunto en $\mathcal{S}$ con un string.

[1] Es interesante resaltar que generar los conjuntos involucra implementar un backtracking, salvo que ya venga implementado en el lenguaje.

▷ El problema es encontrar $S \in \mathcal{S}$ de cardinal máximo tal que ningún par de personas $v_i v_j \in S$ se lleven mal.

▷ Búsqueda exhaustiva: generar todos los subconjuntos de V y evaluar el cárdinal y la relación de llevarse mal. Complejidad $T(n+m) = \Theta(m2^n)$, donde $|V| = n$ y $|E| = m$.

▪ Vacaciones (TRAVELLING SALESMAN): el input es un conjunto $V = \{v_1, \ldots, v_n\}$ de locaciones a visitar (incluyendo el hotel v_1), junto con una matriz D tal que D_{ij} indica el tiempo de viaje de v_i a v_j. El problema es determinar en que orden visitar todas las locaciones exactamente una vez, empezando y terminando en el hotel, a fin de minimizar el tiempo de viaje.

▷ El conjunto V define el conjunto $\mathcal{S}$ de permutaciones de V cuyo primer elemento es v_1. Luego, cada posible recorrido se puede codificar con un string $s_1, \ldots, s_n$ de símbolos todos distintos en $\{1, \ldots, n\}$ ($s_1 = 1$) donde v_i es la j-ésimo locación recorrida si y sólo si $s_j = i$.

▷ El problema es encontrar el recorrido (string) $s(1), \ldots, s(n)$ tal que $\sum_{i=1}^{n} D_{s(i)s(i+1)} + D_{s(n)s(1)}$ sea mínimo.

▷ Búsqueda exhaustiva: generar todas las posibles permutaciones de V y evaluar el costo. Complejidad $T(n^2) = O((n+1)!)$ (notar que D requiere $\Theta(n^2)$ espacio).

1.3. Solución genérica

▪ Supongamos que conocemos el conjunto $\mathcal{S}$ de cadenas sobre Σ a recorrer, y que la misma es "libre de prefijos"[2]. Decimos que una cadena P es una *solución parcial* cuando P es el prefijo de algun $S \in \mathcal{S}$. Todas aquellas soluciones parciales $P' = P + s$ para un símbolo $s \in \Sigma$ son los *sucesores* de P.

▪ Supongamos también que tenemos una función eval que, de alguna forma, permite comparar entre un conjunto de elementos de $\mathcal{S}$, quedándose con el "mejor"[3]. Obviamente, esta noción está hablando de problemas de optimización; para problemas de decisión, alcanza con que eval elija cualquier solución válida.

▪ Podemos definir la siguiente función recursiva mejor, que determina el "mejor" $S \in \mathcal{S}$ que contiene a P como prefijo.

$$\text{mejor}(P) = \begin{cases} P & \text{si } P \text{ pertenece a } \mathcal{S} \\ \text{eval}(\{\text{mejor}(P') \mid P' \text{ es sucesor de } P\}) & \text{caso contrario} \end{cases}$$

▪ La función mejor *define* un algoritmo genérico de backtracking. En general, la descripción de esta función recursiva es suficiente para describir un algoritmo de backtracking. De más esta decir que hay que definir correctamente $\mathcal{S}$ y eval, a partir del input I, para que el algoritmo esté bien definido.

▪ El *árbol de backtracking* tiene un nodo por cada solución parcial, donde P es padre de P' cuando P' es un sucesor de P. Si etiquetamos cada arista con el último símbolo de P', cada P queda descrito como el camino de la raíz a su nodo correspondiente. Notar que el árbol de backtracking corresponde también al árbol de llamadas recursivas.

▪ Para implementar la función mejor en un lenguaje imperativo, la forma más simple es escribiendo directamente la recursión que se muestra a continuación.

▷ La función `esTotal` evalúa si $P \in \mathcal{S}$, mientras que `sucesores` denota el conjunto de todos los sucesores de una solución parcial.

▷ La función `visitar` retorna falso cuando se sabe que ningún $S \in \mathcal{S}$ que tiene a P como prefijo es la solución buscada. La idea es que `visitar` sea rápida y permite *podar* el árbol. Su uso se explica en los ejemplos.

[2] Ningún $S \in \mathcal{S}$ es prefijo propio de otra cadena de $\mathcal{S}$.

[3] Formalmente, suponemos que hay un orden total entre los elementos de $\mathcal{S}$

▷ Remarcamos que este método es un esquema con fines expositivos; no necesariamente indica la mejor forma de implementar cualquier backtracking. La forma de implementar un backtracking depende del problema en cuestión; ver ejemplos.

Algoritmo genérico de backtracking

```
mejor(parcial):
  Si esTotal(parcial): retornar parcial
  Si no visitar(parcial): retornar ⊥
  Retornar eval({mejor(p) para p en sucesores(parcial)})
```

1.4. Adaptando el algoritmo genérico: solución a los ejemplos

1.4.1. 2-partition

- 2-PARTITION generalizado: dado un conjunto $X = \{x_1, \ldots, x_n\}$ de naturales y $k \in \mathbb{Z}$, determinar si X contiene un subconjunto que sume k.
- Sea $\text{part}(X, k)$ la función que indica si X tiene un subconjunto de tamaño k.
- Supongamos que $\text{part}(\{x_1, \ldots, x_n\}, k)$ es verdadera, i.e., existe $Y \subseteq X$ con $\sum Y = k$. Hay dos posibilidades: o bien $x_n \in Y$ o bien $x_n \notin Y$. En el primer caso, $Y \setminus \{x_n\}$ es un subconjunto de $\{x_1, \ldots, x_{n-1}\}$ que suma $k - x_n$; en el segundo caso Y es un subconjunto de $\{x_1, \ldots, x_n\}$ que suma k. Es fácil ver que la recíproca también es verdadera, lo que nos lleva a nuestra definición recursiva:

$$\begin{aligned}\text{part}(\{x_1, \ldots, x_n\}, k) =& (n = k = 0) \vee (n > 0 \wedge \\ & (\text{part}(\{x_1, \ldots, x_{n-1}\}, k - x_n) \vee \text{part}(\{x_1, \ldots, x_{n-1}\}, k)))\end{aligned}$$

- La función part se traduce inmediatamente a una función imperativa. Conviene, sin embargo, evitar el uso de un conjunto como parámetro, cambiandolo por un parametro numérico i que indica que el paramétro considerado en la recursion es $\{x_1, \ldots, x_i\}$.
- La siguiente es una posible implementación de esta función en C++; notar que n está en base 0 y no 1.

Función part, versión 1.0

```
vector<int> X;
bool part(int n, int k) {
  return (n == -1 and k == 0) or (n >= 0 and (part(n-1, k-X[n]) or part(n-1, k)));
}
```

- Complejidad $T(n) = O(2^n)$.
- Notemos que $\text{part}(X, 0)$ es verdadero siempre, mientras que $\text{part}(X, k < 0)$ y $\text{part}(X, k > \sum X)$ son falsos.
- Podemos implementar podas eficientes si mantenemos $\sum X$ como un parámetro auxiliar.

Función part, versión 2.0 (part.cpp)

```
vector<int> X;
bool part(int n, int k, int sum) {
```

```
  return (k == 0) or //primer poda, no importa n
         (k > 0 and n >= 0 and k <= sum and //segunda poda
         (part(n-1, k-X[n], sum-X[n]) or part(n-1, k, sum-X[n])));
}
```

1.4.2. Fiesta (INDEPENDENT SET)

- Problema: dados $V = \{v_1, \ldots, v_n\}$ y una lista E de pares de V, encontrar el conjunto de cardinal máximo que no contiene un par de E.

- Como antes, hay dos posibilidades para v_n; o bien está o no en la solución W. Sin embargo, si v_n está en W, entonces W no puede contener a ningún elemento de $N(v_n) = \{v_i \mid v_i v_n \in E\}$. Con lo cual, tenemos la siguiente función recursiva que determina el cardinal $|W|$ de W.

$$\text{fiesta}(V) = \begin{cases} 0 & \text{si } V = \emptyset \\ \text{máx}(\text{fiesta}(V \setminus \{v\}, E), 1 + \text{fiesta}(V \setminus (N(v) \cup \{v\}), E)) & \text{caso contario} \end{cases}$$

donde v es un elemento cualquier de V

- Implementación en C++ con complejidad: $T(n) = O(m2^n)$.

  ▷ vamos a considerar los posibles invitados en un orden $v_0, \ldots, v_{n-1}$.

  ▷ Mantenemos un índice `v` cuyo valor v representa los invitados recorridos hasta el momento.

  ▷ Además, mantenemos la solución parcial en `vector<int> parcial`, la mejor solución encontrada en `vector<int> mejor`, y un `vector<int> enemistades` que en la posición i tiene la cantidad de conflictos de i con los elementos en la solución parcial. Con esta representación, $v_i \in V$ si $i > v$ y `enemistades[i] = 0`. La ventaja de esta representación es que no necesitamos hacer copias ni de la solución parcial ni del conjunto V.

Problema de la fiesta (fiesta.cpp)

```
vector<vector<int>> N;
vector<int> parcial, mejor, enemistades;

void fiesta(int v) {
  if(v == N.size()) {
    if(parcial.size() > mejor.size()) mejor = parcial;
    return;
  }
  fiesta(v+1); //caso v no está en la solución
  if(enemistades[v] == 0) {
    //caso v está en la solución; este caso se examina sólo si v ∈ V
    for(auto w : N[v]) enemistades[w]++;
    parcial.push_back(v);
    fiesta(v+1);
    parcial.pop_back();
    for(auto w : N[v]) enemistades[w]--;
  }
}
```

1.4.3. Vacaciones (TRAVELLING SALESMAN)

- Problema: dados $V = \{v_1, \ldots, v_n\}$ y $D \in \mathbb{N}^2$, encontrar la permutación $v_{\pi(1)}, \ldots, v_{\pi(n)}$ con $\pi(1) = 1$ que minimize $\sum D_{\pi(i)\pi(i+1)}$

- En este caso, alcanza con elegir qué locación $v \in V \setminus R$ extiende la solución parcial $R = v_{\pi(1)}, \ldots, v_{\pi(|R|)}$.

- Sea $\text{route}(v_i, W)$ el peso de la ruta de peso mínimo que empieza en v_i y termina en v_1, pasando por todos las locaciones de W. Notar que la ruta buscada es $\text{route}(v_1, V \setminus \{v_1\})$. Luego

$$\text{route}(v_i, W) = \begin{cases} D_{i1} & \text{si } W = \emptyset \\ \text{mín}\{\text{route}(v_j, W \setminus v_j) + D_{ij} \mid v_j \in W\} & \text{caso contrario} \end{cases}$$

- Implementación en C++: complejidad $T(n^2) = O((n+1)!)$

  ▷ Mantenemos la solución parcial en `vector<int> parcial`, la mejor solución encontrada hasta el momento en `vector<int> parcial`, su valor en `int mejorcosto` y el conjunto de locaciones ya visitadas en `vector<bool> visitado`, donde v_i ya fue visitada si y sólo si `visitado[i]`.

Problema de las vacaciones (route.cpp)

```
vector<vector<int>> D;
vector<bool> visitado;
vector<int> parcial, mejor;
int mejorcosto;

void route(int from, int cost) {
  if(parcial.size() == D.size() and cost + D[from][0] < mejorcosto)
    {mejorcosto = cost + D[from][0]; mejor = parcial; return;}
  /* if(podar) return: ejercicio */
  for(int v = 1; v < D.size(); ++v) if(not visitado[v]) {
    visitado[v] = true;
    parcial.push_back(v);
    route(v, cost + D[from][v]);
    parcial.pop_back();
    visitado[v] = false;
  }
}
```

1.5. Algunas recomendaciones

- Escribir la función recursiva en forma matemática. No sólo sirve para entender el problema a resolver; en muchos contextos es lo único que se necesita para describir un algoritmo de backtracking.

- Para AED3, alcanza con implementar la recursión en forma directa, a fin de recorrer el árbol de backtracking con la estrategia *primero a lo profundo* (DFS por sus siglas en inglés). En implementaciones reales, otro tipo de recorrido del árbol puede ser conveniente, pero requiere implementar el árbol de alguna forma.

- Evitar copiar recursivamente los argumentos cuando sea posible.

- Las podas incrementan el costo en peor caso, pero evitan el recorrido. Implementar distintas podas lo más eficientemente posible, aplicándolas de acuerdo a su costo y conveniencia.

- Tratar de podar el árbol lo antes posible para recorrer la menor cantidad de soluciones potenciales. ¡El orden en que se recorren los datos importa!

Intervalo (10 mins)

2. Programación dinámica (60 mins)

PD in a nutshell: supongamos que queremos computar una función recursiva f. Si f tiene la propiedad de *superposición de subproblemas*, entonces *memoizar* los resultados para no resolver el mismo problema varias veces.

2.1. Ejemplo motivador: número combinatorio

- El número combinatorio $\binom{n}{k}$ cuenta la cantidad de subconjuntos Y de cardinal k que existen en un conjunto X de cardinal n.

- Si $X = \{x_1, \ldots, x_n\}$, hay dos posibilidades para x_n: o bien $x_n \in Y$ o bien $x_n \notin Y$. En el primer caso, $Y \setminus \{x_n\}$ es un subconjunto de $X \setminus \{x_n\}$ de cardinal $k-1$. En el otro caso, Y es un subconjunto de $X \setminus \{x_n\}$. En consecuencia,

$$\binom{n}{k} = \begin{cases} 1 & \text{si } n = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{caso contrario} \end{cases}$$

- Esta función se puede computar directamente; la instancia (n, k) tiene un costo temporal

$$t(n,k) = \begin{cases} \Theta(1) & \text{si } k = 0 \text{ o } k = n \\ t(n-1,k-1) + t(n-1,k) + \Theta(1) & \text{caso contrario} \end{cases}$$

con lo cual $t(n,k) = \Theta(\binom{n}{k}) = \Theta(n^{\min(k,n-k)})$; a partir de ahora $k < n - k$.

- Sin embargo, cualquier llamada recursiva se hace con una instancia (i, j) donde $0 \le i \le n$ y $0 \le j \le k$. Es decir que se realizan $\Omega(n^k)$ llamadas recursivas, pero sólo sobre $O(nk) = O(n^2)$ subinstancias. En conclusión, algunas subinstancias se tienen que estar ejecutando una cantidad exponencial de veces, i.e., $\binom{n}{k}$ tiene la propiedad de *superposición de subinstancias*.

- En lugar de calcular varias veces el mismo valor, podemos guardarlo en un diccionario M de forma tal que $M[i, j]$ permita acceder al resultado de la instancia (i, j) cuando sea necesario. A esta estrategia, íntimamente relacionada con la idea de cache de una computadora, se la llama *memoización*.

- La forma simple de implementar esta estrategia es: 1. escribir el algoritmo recursivo, 2. agregar el código para memoizar, i.e., guardar cada valor la primer vez que se computa y luego evitar recalcular las instancias ya resueltas.

Paso 1: escribir la función recursiva

```
using ull = unsigned long long;
ull choose(ull n, ull k) {
  return k == 0 or k == n ? 1 : choose(n-1,k) + choose(n-1, k-1);
}
```

Paso 2: agregar codigo de memoización (choose.cpp)

```
const int undefined = -1;
vector<vector<int>> M;

ull choose(ull n, ull k) {
```

```
5    if(k == 0 or k == n) return 1;
6    if(M[n][k] == undefined) //calcular una vez y guardar en cache
7      M[n][k] = choose(n-1, k-1) + choose(n-1, k);
8    return M[n][k];
9  }
```

- Un tema no menor es cómo se representa la estructura de memoización M. En este caso, alcanza con utilizar una matriz de $n \times k$, ya que permite cargar y acceder a cada valor en $O(1)$ tiempo.

2.2. Algoritmo genérico

- Queremos computar una función recursiva $\text{PD}(x) = f(\text{PD}(x_1), \ldots, \text{PD}(x_k))$, para alguna función f, tal que:

  superposición de subproblemas: $\text{PD}(x)$ se invoca recursivamente sobre una cantidad s posibles de subinstancias y, sin embargo, se requieren $\omega(s)$ llamadas recursivas para computar $\text{PD}(x)$.

- La solución es **memoizar**: guardar un diccionario para almacenar la solución a cada subinstancia. Antes de resolver una subinstancia, acceder al diccionario para ver si esta definida.

- Algoritmo genérico *top-down*: como antes, recordar que es sólo un esquema y no una receta.

  ▷ La función b computa directamente el resultado sobre x y se usa para el caso base.

Algoritmo genérico de programación dinámica

```
Sea M = ∅ un diccionario de instancias en soluciones (M[x] → PD(x))
pd(x):
  Si x es un caso base, retornar b(x)
  Si M[x] = ⊥, definir M[x] = f(pd(x_1), ..., pd(x_n))
  retornar M[X]
```

2.3. Solución a otros problemas

2.3.1. 2-partition

- Recordemos que:

$$\begin{aligned}\text{part}(\{x_1, \ldots, x_n\}, k) =& (n = k = 0) \vee (n > 0 \wedge \\ & (\text{part}(\{x_1, \ldots, x_{n-1}\}, k - x_n) \vee \text{part}(\{x_1, \ldots, x_{n-1}\}, k)))\end{aligned}$$

- Notar que toda subinstancia es de la forma $\{x_1, \ldots, x_i\}$, con lo cual en lugar de usar un conjunto como parámetro de part, podemos usar el índice $0 \leq i \leq n$.

$$\text{part}(n, k) = (n = k = 0) \vee (n > 0 \wedge (\text{part}(n-1, k - x_n) \vee \text{part}(n-1, k)))$$

- En consecuencia, la cantidad de posibles instancias es $O(nk)$, mientras que la cantidad de llamadas recursivas es $O(2^n)$.

- Cuando $k \ll 2^n$, hay superposición de subproblemas. Cuando $k > 2^n$, no vale la pena guardar una estructura de memoización (ver abajo).

- Dos opciones para la estructura de memoización:

  Matriz: porque las instancias se acceden con indices. Hay que inicializarla, con lo cual la instancia (n, k) tiene un costo $t(nk) = \Theta(nk)$.

  Hashing: porque permite acceso en $O(1)$ esperado. A pesar de ser más lenta que la matriz, no hay que inicializarla, con lo cual el costo "baja" de $\Theta(nk)$ a $\Theta(nh) = O(n2^n)$ esperado, donde h es la cantidad de subinstancias consultadas. Claro que esto solo conviene si k es muy grande y no parece valer la pena para este tipo de ejercicios.

- La implementación en C++ con una matriz es una aplicación directa del esquema.

Función part, versión 3.0 (part_pd.cpp)

```
vector<int> X;
vector<vector<int>> M;
const int undefined = -1;
//unordered_map<pair<int,int>, bool> M; //otra opcion: ejercicio

bool part(int n, int k) {
  if(k == 0) return 1;
  if(k < 0 or n < 0) return 0;
  if(M[n][k] == undefined)
    M[n][k] = part(n-1, k-X[n]) or part(n-1, k);

  return M[n][k];
}
```

- La complejidad del algoritmo resultante, donde x es la cantidad de bits de k, es:

  Matriz: $T(n + x) = O(n2^x)$.

  Hashing: $T(n + x) = O(n2^{\min\{n,x\}})$ esperado.

- Comparar con $T(n + x) = O(2^n)$ del algoritmo de backtracking.

2.3.2. Construyendo una solución

- Supongamos que, en lugar de querer saber si existe un subcojunto de X que sume k, queremos encontrar uno de estos subconjuntos Y si existe.

- Podemos aprovechar la definición de part con las siguientes observaciones:

  ▷ Y existe si y sólo si part(n, k) es verdadero.

  ▷ Por definición, Y existe si y sólo si part$(n - 1, k - x_n)$ o part$(n - 1, k)$ es verdadero.

  ▷ En el primer caso, existe solución Y que contiene a x_n, en el segundo ninguna solución Y tiene a x_n.

- El algoritmo resultante se aprovecha de la función part ya implementada para encontrar la solución.

- La complejidad es $O(n)$ adicional al costo de part.

Obtención de un subcojunto que suma k (part_pd.cpp)

```
vector<int> get_part(n,k) {
  assert(k >= 0 and n >= 0);
  return k == 0 ? vector<int>() : (part(n-1,k) ? get_part(n-1, k) : get_part(n-1,k-X[n]) +
  ↪  X[n]);
}
```

2.3.3. Vacaciones (TRAVELLING SALESMAN)

- Recordar que:

$$\text{route}(v_i, W) = \begin{cases} D_{i1} & \text{si } W = \emptyset \\ \text{mín}\{\text{route}(v_j, W \setminus v_j) + D_{ij} \mid v_j \in W\} & \text{caso contrario} \end{cases}$$

- En este caso, la cantidad de subinstancias es $O(n2^n)$, ya que consiste en todas las posibilidades de combinaciones de $v_i \times W$.

- La cantidad de llamadas recursivas es $\Omega(n!)$, en consecuencia hay superposición de subproblemas.

- La estructura de memoización queda como ejercicio para aquellos que gusten de la programación competitiva (consultar e.g. [3]).

- El algoritmo resultante consume una cantidad exponencial de memoria y no es útil en la práctica.

2.4. Limites a la programación dinámica

- Consideremos el problema de la fiesta nuevamente:

$$\text{fiesta}(V) = \begin{cases} 0 & \text{si } V = \emptyset \\ \text{máx}(\text{fiesta}(V \setminus \{v\}, E), 1 + \text{fiesta}(V \setminus (N(v) \cup \{v\}), E)) & \text{caso contario} \end{cases}$$

- La cantidad de llamadas recursivas es $O(2^n)$.

- Notemos que, a priori, cualquier subconjunto de V puede ser usado como parámetro, ya que depende de $N(v)$. En consecuencia, la cantidad de subinstancias posibles es $\Omega(2^n)$. Ergo, esta función recursiva no tiene la propiedad de superposición de subproblemas.

- Esto significa que no vamos a obtener un mejor algoritmo en peor caso usando memoización.

- ¿Significa que no se puede resolver el problema con PD? No a priori; sólo significa que esta función no es apropiada.

- ¿Significa que no vale la pena usar memoización en ningún caso? No a priori: quiza en alguna instancia particular, la cantidad de subinstancias sea mucho menor a 2^n; sin embargo, el algoritmo resultante no sería eficiente en la mayoría de los casos, sino en algunos particulares. (Igual, esto se puede explotar en la práctica, pero no lo vemos en la materia.)

2.5. Algunas consideraciones particulares

- La función recursiva determina si se puede o no aplicar PD.
- Que una función recursiva no permita aplicar PD no significa que ninguna lo permita. Encontrar la función recursiva es la parte difícil. Vemos funciones más complejas en la practica y laboratorio y más adelante en la teórica.
- La implementación recursiva que vimos es *top-down*; existen soluciones iterativas *bottom-up* que permiten reducir el consumo de memoria. Se ve más de esto en la práctica y el laboratorio.
- La dificultad de diseñar un algoritmo PD debería estar en encontrar la función recursiva, no en entender la técnica: recursión + superposición + memoización.

Intervalo (10 mins)

3. Algoritmos golosos (40 mins)

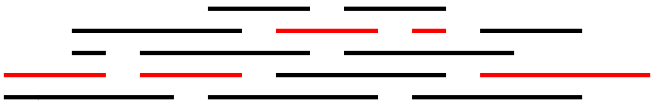
- Tenemos un problema que se puede resolver tomando distintas decisiones en cada paso (donde el resultado de una decisión afecta a las decisiones subsiguientes).
- Algoritmo goloso: tomar la mejor decisión para este paso, sin importar lo que ocurre con los siguientes.
- Es más eficiente que los algoritmos de backtracking y PD donde se consideran todas las posibles decisiones.
- Es más difícil demostrar que el algoritmo es correcto, ya que no se consideran todas las posibles decisiones y por ende hay que ver que la que se toma lleva a la solución.
- Es más difícil diseñar un algoritmo goloso, ya que no sabemos a priori cómo tomar las decisiones.
- En problemas de optimización se puede tomar como *heurística* para obtener una solución aunque no sea óptima.

3.1. Ejemplo: Interval scheduling

- El input es un conjunto de tareas $\mathcal{I} = I_1, \ldots, I_n$, donde cada tarea esta determinada por su comienzo s_i y su final t_i. El problema consiste en seleccionar un conjunto de tareas que no se superpongan y tenga máxima cardinalidad. Formalmente, encontrar $\mathcal{S} \subseteq \mathcal{I}$ de cardinalidad máxima tal que $I \cap I' = \emptyset$ para todo par $I, I' \in \mathcal{S}$.

Ejemplo del problema de interval scheduling

Dados los siguientes intervalos que representan las tareas de acuerdo a su momento de inicio y finalización, un output al problema de interval scheduling está dado por los intervalos rojos. La solución óptima no es única.



- Algoritmo de backtracking: la tarea I_i puede formar o no parte de la solución. Si I_i está en la solución, no puede estar ninguna tarea que la interseque.

- Es igual al problema de la Fiesta (INDEPENDENT SET) bajo un input particular (formado por intervalos). ¿Podemos mejorar el algoritmo de backtracking?

- En lugar de comprobar las dos posibilidades para cada intervalo, consideramos sólo una. Esto es lo mismo que decir que el output se consigue agregando de a una tarea I en cada paso y descartando las tareas que intersecan a I. Cuatro posibles estrategias golosas que podrían o no funcionar:

1. I es la tarea que empieza lo antes posible.
2. I es la tarea que termina lo antes posible.
3. I es la tarea que empieza lo mas tarde posible.
4. I es la tarea que termina lo más tarde posible

- Las estrategias simétricas 2 y 3 llevan a la solución óptima, como se demuestra en el siguiente teorema.

- Digamos que una subfamilia $\mathcal{S}$ de $\mathcal{I}$ es *optima* cuando todos los intervalos de $\mathcal{S}$ son mutuamente disjuntos y $|\mathcal{S}| \leq |\mathcal{I}'|$ para toda subfamilia $\mathcal{I}'$ de $\mathcal{I}$ cuyos intervalos son mutuamente disjuntos.

Teorema 1. *Sea $\mathcal{I} = \{I_1, \ldots, I_n\}$ una familia de intervalos con $I_i = [s_i, t_i]$. Si $\mathcal{S} = I_{i(1)}, \ldots, I_{i(k)}$ es una subfamilia tal que: 1. $I_{i(1)}$ es un intervalo con $t_{i(1)}$ mínimo, 2. para $1 \leq j < k$, $I_{i(j+1)}$ es un intervalo con $t_{i(j+1)}$ mínimo de entre aquellos con $s_{i(j+1)} > t_{i(j)}$ y 3. no existe un intervalo I_i con $s_i > t_{i(k)}$, entonces $\mathcal{S}$ es óptimo.*

Demostración. Vamos a mostrar por inducción en j que alguna solución óptima contiene a $\mathcal{S}_j = I_{i(1)}, \ldots, I_{i(j)}$ para todo $0 \leq j \leq k$. Luego, como todos los intervalos intersecan algún intervalo de $\mathcal{S} = \mathcal{S}_k$ por 1–3, concluimos que $\mathcal{S}$ es óptima. El caso base $j = 0$ es trivial porque $\mathcal{S}_0 = \emptyset$. Para el caso $j \Rightarrow j + 1$ con $j < k$, consideremos cualquier solución óptima $\mathcal{S}_j \cup \{I_p\} \cup \mathcal{J}$ tal que $t_p < t_h$ para todo $I_h \in \mathcal{J}$. Notar que esta solución óptima existe por hipótesis inductiva. Más aún, como I_p no interseca a ninguna tarea de $\mathcal{S}_j$, tenemos $t_p \geq t_{i(j+1)}$ por 1 y 2. Más aún, $s_p \geq t_{i(j+1)}$ porque I_p no interseca a $t_{i(j+1)}$. En consecuencia, por 2, $t_{i(j+1)} \leq t_p < s_h$ para todo $I_h \in \mathcal{J}$ y, por lo tanto, $\mathcal{S}_j \cup \{I_{i(j+1)}\} \cup \mathcal{J}$ también es solución óptima. □

- Implementación del algoritmo: ordenar las tareas por tiempo de finalización t. Luego, recorrerlas en este orden, manteniendo el tiempo t de finalización de la última tarea seleccionada. Si la tarea revisada empieza mas tarde que t, seleccionarla y actualizar t.

- Código en C++: los intervalos se especifican con un par de vectores que se pasan como parámetro. Los valores s_i y t_i del i-ésimo intervalo están determinados por `S[i]` y `S[i]`, respectivamente. La variable `int last_T` mantiene el valor de t, mientras que `vector<int> res` contiene los índices de los intervalos de la solución golosa.

- En esta implementación suponemos que $t_1 < \ldots < t_n$ por simplicidad (sino, hay que ordenar).

Algoritmos para interval scheduling (sched.cpp)

```
vector<int> sched(const vector<int>& S, const vector<int>& T) {
  int last_T = -1;
  vector<int> res;
  for(int i = 0; i < S.size(); ++i) {
    if(S[i] >= last_T) {
      res.push_back(i);
      last_T = T[i];
    }
  }
  return res;
```

```
11 }
```

- Complejidad: $T(n) = O(n) + T(\text{sort}(n))$, donde $\text{sort}(n)$ es el costo de ordenar el input (0 si ya viene ordenado, $O(n)$ si los números son chicos, $O(n \log n)$ en el caso general).

3.2. Heurísticas golosas

- Una heurística es un algoritmo para un problema de optimización que retorna una solución factible aunque no necesariamente óptima.

- Las heurísticas constructivas golosas son heurísticas que aplican un algoritmo goloso.

- Ejemplo para el problema de la fiesta: en cada paso invitar a la persona que menos conflictos tenga. Complejidad: $T(n + m) = O(n + m)$ ¿Siempre encuentra la solución óptima?

- Ejemplo para el problema de las vacaciones: en cada paso visitar la locación más cercana. Complejidad: $T(n^2) = O(n^2)$ ¿Siempre encuentra la solución óptima?

4. Comentarios bibliográficos

Los temas de esta clase se pueden estudiar de [5, Capítulos 7 y 8], [1, Capítulos 6–9], o [2, Capítulos 15 y 16]. Alcanza con leer cada tema de cualquiera de estos libros, que forman parte de la bibliografía obligatoria. Sin embargo, todos ellos tienen una gran cantidad de información y prerrequisitos (e.g., grafos), lo que puede resultar desafiante cuando uno quiere 1. hacer una primera aproximación al tema o 2. entender cómo implementar los algoritmos en la práctica. En estos casos, se recomienda consultar un libro de programación competitiva, particularmente [4, Capítulos 5–7, versión web].

Referencias

[1] Gilles Brassard and Paul Bratley. *Fundamentals of algorithmics*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1996.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, third edition, 2009.

[3] Steven Halim and Felix Halim. *Competitive Programming*. Self published, 3rd edition, 2013.

[4] Antti Laaksonen. *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*. Springer, 2017.

[5] Steven S. Skiena. *The Algorithm Design Manual*. Springer, second edition, 2008.