Técnicas Algorítmicas

Prof. Agustín Gravano

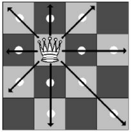
MiM - UTDT - Segundo semestre de 2020

Clase 1: Recursión algorítmica

Técnicas Algorítmicas

Técnicas de programación para encarar problemas no triviales.

- Ordenar una lista `L` en $O(n \log n)$, donde $n =$ `len(L)`.
- Determinar la menor cantidad de ediciones (ins/sust/elim) necesarias para llegar de una palabra A a otra palabra B. Ejemplo: `naranja` $\rightarrow$ `manzana` requiere 4 ediciones
- Colocar 8 reinas de ajedrez en un tablero, sin que se amenacen entre sí.



- Encontrar el camino más corto que pase una vez por cada ciudad en el mapa y vuelva a la ciudad de origen.

Técnicas Algorítmicas

Clase 1: Recursión
Clase 2: Dividir y conquistar
Clase 3: Programación dinámica
Clase 4: Backtracking
Clase 5: Heurísticas

Cronograma:

ma 13/10: Clase 1. Simulacro de evaluación
ma 20/10: Evaluación de Clase 1 (30'). Clase 2
ma 27/10: Evaluación de Clase 2 (30'). Clase 3
ma 3/11: Eval. de Clase 3 (30'). Clase 4. Presentación TP
ma 10/11: Clase 5. Presentación TP

Ver el criterio de evaluación en la página de la materia.

Técnicas Algorítmicas

Prerrequisitos:

- Python 3
- (Recomendado) Anaconda 3.X
- Dominar los temas de "Fundamentos de Algoritmos".
- Tener acceso al campus.

Por cualquier inconveniente, por favor consultar hoy mismo durante el recreo.

Hoy haremos el simulacro de evaluación individual justo después del recreo (cerca de las 20:45).

Ejemplo: Factorial

Problema: Dado un entero $n \geq 0$, devolver $n! = \prod_{i=1}^{n} i$

```
def fact(n):
  res = 1
  i = 1
  while i <= n:
    res = res * i
    i = i + 1
  return res
```

Este algoritmo es iterativo y tiene complejidad lineal: $O(n)$.

Ejemplo: Factorial

Observación: $$\begin{aligned} \mathtt{fact}(n) &= \prod_{i=1}^{n} i \\ &= \big(1 \cdot 2 \cdot \ldots \cdot (n-1)\big) \cdot n \\ &= \mathtt{fact}(n-1) \qquad\quad \cdot n \end{aligned}$$

$\mathtt{fact}(n) = \mathtt{fact}(n-1) \cdot n$ es una **definición recursiva**, pero todavía está incompleta.

```
1 def fact(n):
2   res = fact(n-1) * n
3   return res
```

Ejemplo: queremos ejecutar `fact(3)`, pero esto requiere ejecutar `fact(2)` (en la línea 2), que a su vez requiere ejecutar `fact(1)`, que a su vez requiere ejecutar `fact(0)`, que a su vez requiere ejecutar `fact(-1)`, que a su vez...

Ejemplo: Factorial

Nos faltó frenar la recursión: cuando llegamos a `fact(0)`, ya podemos parar y devolver el resultado (1). A esto se lo llama caso base.

```
def fact(n):
  if n==0:    # caso base
    res = 1
  else:       # caso recursivo
    res = fact(n-1) * n
  return res
```

Ejemplo: queremos ejecutar `fact(3)`, pero esto requiere ejecutar `fact(2)`, que a su vez requiere ejecutar `fact(1)`, que a su vez requiere ejecutar `fact(0)`, que devuelve 1, lo cual frena la recursión. Después, *a la vuelta de la recursión*, `fact(1)` devuelve 1; `fact(2)` devuelve 2 y por último `fact(3)` devuelve 6.

Ejemplo: Factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

$$n! = \prod_{i=1}^{n} i$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ (n-1)! \cdot n & \text{si } n > 0 \end{cases}$$

```
# alg iterativo
def fact(n):
  res = 1
  i = 1
  while i <= n:
    res = res * i
    i = i + 1
  return res
```

```
# alg recursivo
def fact(n):
  if n==0:
    res = 1
  else:
    res = fact(n-1) * n
  return res
```

Recursión algorítmica

La solución a un problema depende de la solución a instancias de menor tamaño del mismo problema.

Recursión algorítmica

1. Resolver el problema para los casos base.

2. Suponiendo que se tiene resuelto el problema para instancias de menor tamaño, modificar dichas soluciones para obtener una solución al problema original.

La recursión ofrece otra forma de ciclar o repetir código.

Ejercicios para resolver ahora

1. Escribir una función recursiva `suma` que, dado un entero $n \geq 0$, devuelva $\sum_{i=1}^{n} i$

En la página de la materia, sección "Ejemplos", está el código de la función recursiva de factorial, que pueden usar como referencia.

Ejemplo: Sumar elementos de una lista

Problema: Dada una lista de números, devolver la suma.

Algoritmo iterativo:

```
def sumar(L):
  res = 0
  i = 0
  while i<len(L):
    res = res + L[i]
    i = i + 1
  return res
```

Ejemplo: Sumar elementos de una lista

Observación: `sumar([4,1,3,7]) = 4 + sumar([1,3,7])`

En general, podemos definir `sumar` en forma recursiva:

$$\texttt{sumar(L)} = \begin{cases} \texttt{0} & \text{si } \texttt{L} \text{ es vacía} \\ \texttt{L[0] + sumar(L[1:])} & \text{si no} \end{cases}$$

```
def sumar(L):
  if len(L) == 0:
    res = 0
  else:
    res = L[0] + sumar(L[1:])
  return res
```

Notar que `sumar(L[1:])` es un problema menos complejo que `sumar(L)`, y está más cerca del caso base `sumar([])`.

Ejercicios para resolver ahora

1. Escribir una función recursiva `suma` que, dado un entero $n \geq 0$, devuelva $\sum_{i=1}^{n} i$
2. Escribir una función recursiva `maximo` que, dada una lista no vacía de números, devuelva el número máximo.

En la página de la materia, sección "Ejemplos", está el código de una función recursiva que suma los elementos de una lista (función `sumarRec1`), que pueden usar como referencia.

Complejidad algorítmica

```
def fact(n):
  if n==0:    # caso base
    res = 1
  else:       # caso recursivo
    res = fact(n-1) * n
  return res
```

$$\begin{aligned}\texttt{fact(4)} &= \texttt{fact(3)} * 4\\ &= \texttt{fact(2)} * 3 * 4\\ &= \texttt{fact(1)} * 2 * 3 * 4\\ &= \texttt{fact(0)} * 1 * 2 * 3 * 4\\ &= 1 * 1 * 2 * 3 * 4 = 24\end{aligned}$$

`fact(`n`)` llama recursivamente n veces a la función `fact`, con n cada vez menor. En cada ejecución de `fact`, además del llamado recursivo hay solo instrucciones $O(1)$. Luego, como `fact(`n`)` hace n veces algo $O(1)$, su complejidad es $O(n) \times O(1) = O(n)$ (lineal).

Complejidad algorítmica

```
def sumar(L):
  if len(L) == 0:
    res = 0
  else:
    res = L[0] + sumar(L[1:])
  return res
```

$$
\begin{aligned}
\texttt{sumar([4,1,3,7])} &= 4 + \texttt{sumar([1,3,7])} \\
&= 4 + 1 + \texttt{sumar([3,7])} \\
&= 4 + 1 + 3 + \texttt{sumar([7])} \\
&= 4 + 1 + 3 + 7 + \texttt{sumar([])} \\
&= 4 + 1 + 3 + 7 + 0 = 15
\end{aligned}
$$

Para una lista `L`, `sumar(L)` llama `len(L)` veces a la función `sumar`, pasándole cada vez una lista de menor tamaño.

En cada ejecución de `sumar`, además del llamado recursivo hay varias instrucciones $O(1)$ y una de orden **lineal**: `L[1:]`.

En consecuencia, la complejidad de `sumar(L)` es $O(\texttt{len(L)}) \times O(\texttt{len(L)}) = O(\texttt{len(L)}^2)$; es decir, cuadrática.

Complejidad algorítmica

Veamos entonces un algoritmo recursivo y lineal. Agregamos un segundo parámetro para indicar desde qué posición hay que sumar. Así evitamos copiar la lista en cada llamado recursivo.

```
def sumar_aux(L, desde):
  if desde >= len(L):
    res = 0
  else:
    res = L[desde] + sumar_aux(L, desde+1)
  return res
```

Por último, sólo falta que la función `sumar` llame a la auxiliar:

```
def sumar(L):
  return sumar_aux(L, 0)
```

A esta técnica se la conoce como inmersión de parámetros.

Complejidad algorítmica

```
L = [4,1,3,7]
```

$$
\begin{aligned}
\texttt{sumar(L)} &= \texttt{sumar\_aux(L, 0)} \\
&= 4 + \texttt{sumar\_aux(L, 1)} \\
&= 4 + 1 + \texttt{sumar\_aux(L, 2)} \\
&= 4 + 1 + 3 + \texttt{sumar\_aux(L, 3)} \\
&= 4 + 1 + 3 + 7 + \texttt{sumar\_aux(L, 4)} \\
&= 4 + 1 + 3 + 7 + 0 = 15
\end{aligned}
$$

Para una lista `L`, `sumar(L)` llama $\sim$ `len(L)` veces a la función `sumar_aux`, pasándole siempre la misma lista, pero el valor de `desde` está cada vez más cerca de `len(L)`.

En cada ejecución de `sumar_aux`, además del llamado recursivo hay solo instrucciones $O(1)$. Ergo, la complejidad de `sumar(L)` es $O(\texttt{len(L)}) \times O(1) = O(\texttt{len(L)})$; o sea, lineal.

Ejercicios para resolver ahora

1. Escribir una función recursiva `suma` que, dado un entero $n \geq 0$, devuelva $\sum_{i=1}^{n} i$
2. Escribir una función recursiva `maximo` que, dada una lista no vacía de números, devuelva el número máximo.
3. Analizar la complejidad de las funciones recursivas `suma` y `maximo`.
4. Si `maximo` es cuadrática, modificarla para que sea lineal.

Repaso de la clase de hoy

- Recursión algorítmica
- Caso base y caso recursivo de las funciones recursivas
- Complejidad algorítmica de funciones recursivas
- Técnica de inmersión de parámetros para evitar copias innecesarias de estructuras complejas

Con lo visto hoy, ya pueden resolver la sección 1 de la guía de ejercicios.

Para la clase próxima, repasar selection sort (ver siguientes diapos).

Selection sort

Para cada `i` entre 0 y `len(A)`-1 (inclusive):

Buscar el menor elemento en `A[i:]`.

Intercambiarlo con `A[i]`.

0	1	2	3	4	5	6	7
59	7	388	41	2	280	50	123
2	7	388	41	59	280	50	123
2	7	388	41	59	280	50	123
2	7	41	388	59	280	50	123
2	7	41	50	59	280	388	123
2	7	41	50	59	280	388	123
2	7	41	50	59	123	388	280
2	7	41	50	59	123	280	388
2	7	41	50	59	123	280	388

Propiedad invariante: `A[0:i]` está ordenada.

```
def pos_minimo(L):
  minimo = 0                  O(1)
  for j in range(0, len(L)):  len(L) iteraciones
    if L[j] < L[minimo]:      O(1)
      minimo = j              O(1)
  return minimo               O(1)
```

Esto ejecuta `len(L)` veces operaciones de orden constante.
Entonces, `pos_minimo(L)` es lineal respecto de `len(L)`.

```
def selection_sort(A):
  for i in range(0, len(A)):     O(len(A)) iters.
    min = pos_minimo(A[i:]) + i  O(len(A))
    aux = A[i]                   O(1)
    A[i] = A[min]                O(1)
    A[min] = aux                 O(1)
```

Esto ejecuta `len(A)` veces operaciones que tienen `O(len(A))`.
Entonces, `selection_sort(A)` es cuadrático respecto de `len(A)`.