Algoritmos de ordenamiento

Capítulo 4 — Análisis de Algoritmos con Julia

Eric S. Téllez

Tabla de contenidos

1. Algoritmos de ordenamiento

1.1. Introducción

En este tema se aborda el ordenamiento basado en comparación, esto es, existe un operador $<$ que es capaz de distinguir si un elemento a es menor que un elemento b.

El operador cumple con las siguientes propiedades:

- si $a < b$ y $b < c$ entonces $a < c$ (transitividad); e.g., $1 < 10$ y $10 < 100$ entonces $1 < 100$.
- tricotomía:
  - si $a < b$ es falso y $b < a$ es falso, entonces $a = b$ (antisimetría); dicho de otras formas:
    - si a no es menor que b ni b menor que a entonces a es igual a b,
    - sustituyendo variables, $1 < 1$ es falso, el intercambio es obvio, entonces $1 = 1$.
  - en otro caso, $a < b$ o $b < a$.

[1]

Sin pérdida de generalidad, podemos plantear el problema de ordenamiento sin permitir repeticiones como sigue: dado un arreglo $A[1, n] = a_1, a_2, \cdots, a_n$; un algoritmo de ordenamiento obtiene la permutación π tal que $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$.

[2]

En términos prácticos, la idea es reorganizar A, mediante el cálculo implícito de la permutación π, de tal forma que después de terminar el proceso de ordenamiento se obtenga que A está ordenado, i.e., $a_i \leq a_{i+1}$. En sistemas reales, el alojar memoria para realizar el ordenamiento implica costos adicionales, y es por esto que muchas veces se busca modificar directamente A.

[3]

1.1.1. Costo del problema

Para una entrada de tamaño n existen $n!$ permutaciones posibles; cada una de estas permutaciones es una instancia del problema de ordenamiento de tamaño n.

Existe una permutación objetivo π^*, i.e., que cumple con la definición de que está ordenada; ahora pensemos en un grafo donde cada π_i está conectada con todas las permutaciones en las que se puede transformar haciendo una única operación, e.g., intercambiando un elemento. El algoritmo forma ese grafo con sus posibles decisiones, por lo que el camino más largo *i.e., ruta sin ciclos*, entre cualquier π_i y la permutación π^* es el costo de peor caso del algoritmo.

Ahora, cada operación que realicemos en un algoritmo nos acercará más a π^*, descartando una cierta cantidad de instancias posibles pero no viables; si nuestra función de transición en el grafo viene dada con respecto a colocar cada par de elementos en su orden relativo, entonces, la mitad de las permutaciones se han descartado, ya que ese par no puede estar en el orden contrario. Por tanto, el costo de cualquier algoritmo que realice comparaciones y descarte la mitad del espacio de búsqueda, es $\log_2(n!)$, que usando la aproximación de Stirling,[4] lo podemos reescribir como sigue:

$$\log_2(n!) = n \log_2 n - n \log_2 e + O(\log_2 n)$$

Esto se puede simplemente escribir como $O(n \log n)$.

[1] Usar un operador como $<$ es suficiente para crear algoritmos correctos y eficientes, sin embargo, en la práctica y en una computadora real, también es válido utilizar operadores como $=$ o $\leq$, o intercambiar por $>$ y $\geq$ según convenga. No hay impacto en la eficiencia.

[2] Cuando se permiten elementos repetidos, se le llama *ordenamiento estable* y se asegura que en el arreglo ordenado se preserve el orden posicional original cuando $a = b$. Esta propiedad es importante cuando hay datos satelitales asociados a la llave de comparación.

[3] Utilizar π solo es necesario cuando no es posible modificar A. También es muy común utilizar datos *satélite* asociados con los valores a comparar, de esta manera es posible ordenar diversos tipos de datos. Un ejemplo de esto es ordenar un *dataframe*, pero también estructuras de datos donde existe un campo especial y el resto de los datos asociados es de importancia para una aplicación.

[4] Aproximación de Stirling https://en.wikipedia.org/wiki/Stirling%27s_approximation.

1.2. Algoritmos de ordenamiento

Existen muchos algoritmos que pueden resolver el problema de ordenamiento, es común contar el número de comparaciones ya que produce la información necesaria para la navegación en el grafo de instancias; también es común contar las operaciones de intercambio de elementos. Las pruebas y la navegación en el grafo determinan el costo del algoritmo. Es necesario mencionar que mover datos entre diferentes zonas de memoria puede llegar a ser más costoso que solo acceder a esas zonas, por lo que hay una asimetría en el costo de estas dos operaciones.

Note que algunos de los algoritmos más simples pueden tener un comportamiento oportunista y que son capaces de obtener ventaja en instancias sencillas, por lo que no debería saltarse esas secciones si solo conoce su comportamiento en peor caso.

1.2.1. Bubble sort

El algoritmo de ordenamiento de burbuja o *bubble sort* realiza una gran cantidad de comparaciones, como puede verse en Listado 1, el algoritmo usa dos ciclos anidados para realizar una comparación y una posible transposición, formando un *triángulo*, i.e.,

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-i} O(1);$$

por lo tanto su costo está dominado por el triángulo formado, i.e., $\sim n^2/2$ lo que puede escribirse simplemente como $O(n^2)$.

Listado 1 Bubble sort de peor caso

```
function bubble_sort!(A)
  n = length(A)
  for i in 1:n-1                                    ①
    for j in 1:n-i                                  ②
      if A[j] > A[j+1]
        A[j], A[j+1] = A[j+1], A[j]                 ③
      end
    end
  end

  A
end

bubble_sort!([8, 4, 3, 1, 6, 5, 2, 7])
```

① Ciclo que recorre $n-1$ veces todo el arreglo; y pone el elemento máximo en su posición final.
② Ciclo que recorre $n-i$ veces el arreglo; ya que en cada corrida se pone el máximo en su posición.
③ Intercambio cuando hay pares en desorden.

```
8-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
 7
 8
```

El algoritmo mostrado en Listado 1 es un algoritmo de peor caso, ya que sin importar la complejidad de la instancia (i.e., qué tan alejada está π_i de π^*), se comporta igual.

Es relativamente fácil hacer un bubble sort que tenga en cuenta la complejidad de la instancia, medida como el número de intercambios necesarios.

Listado 2 Bubble sort adaptable

```
function adaptive_bubble_sort!(A)
  n = length(A)

  for i in 1:n-1
    s = 0
    for j in 1:n-i                                          ①
      if A[j] > A[j+1]
        s += 1                                              ②
        A[j], A[j+1] = A[j+1], A[j]
      end
    end
    s == 0 && break                                         ③
  end

  A
end

adaptive_bubble_sort!([7, 8, 4, 3, 1, 6, 5, 2])
```

① La idea es que si no hay intercambios en una iteración, entonces el arreglo ya está ordenado.
② Contador de intercambios.
③ Condición de paro, i.e., no hubo intercambios.

```
8-element Vector{Int64}:
 1
 2
 3
```

```
4
5
6
7
8
```

En la forma Listado 2, bubble sort es capaz de terminar en $n-1$ comparaciones si el arreglo está ordenado, sacando provecho de casos simples en términos de instancias casi ordenadas.

1.2.2. Insertion sort

El algoritmo de ordenamiento por inserción o *insertion sort* es un algoritmo simple que al igual que bubble sort tiene un mal peor caso y puede aprovechar casos simples.

Listado 3 Algoritmo *insertion sort*

```
function insertion_sort!(A)
  n = length(A)
  for i in 2:n                                                    ①
    key = A[i]                                                    ②
    j = i - 1
    while j >= 1 && A[j] > key                                    ③
      A[j + 1] = A[j]                                             ④
      j -= 1
    end

    A[j + 1] = key                                                ⑤
  end

  A
end

insertion_sort!([5, 1, 4, 8, 2, 6, 3, 7])
```

① El algoritmo comienza en la segunda posición del arreglo y revisará todos los elementos.
② Es importante hacer una copia de *key* para simplificar la implementación.
③ La idea general es ordenar las posiciones de $1..i$, para esto se debe recorrer hacia atrás el arreglo completo, para determinar la posición de inserción de *key*.
④ Intercambio de elementos para colocar *key* en su lugar ordenado.
⑤ *key* se pone en su lugar final.

```
8-element Vector{Int64}:
 1
 2
```

```
3
4
5
6
7
8
```

Para analizar Listado 3, es importante notar que el ciclo más externo termina con el subarreglo $A[1..i]$ ordenado; por lo que cuando se comienza el ciclo, si *key* se prueba estar en su posición correcta, entonces ya no es necesario revisar el resto del subarreglo, esto determina que un arreglo ordenado tendrá un costo de $O(n)$ comparaciones; si está *casi ordenado* en términos del número de intercambios necesarios, entonces, el algoritmo se adaptará sacando provecho de la instancia.

En el *peor caso* de insertion sort, el algoritmo no puede parar de manera prematura, e.g., un arreglo en orden reverso, el ciclo `for` se ejecutará $n-1$ veces, mientras que el ciclo `while` deberá revisar el subarreglo completo en cada iteración, sumando un costo de i operaciones en cada iteración, i.e., $\sum_{i=1}^{n} i$, esta forma produce un *triángulo*, resultando en un costo $O(n^2)$.

1.2.3. Quick sort

Quick sort (ver Cormen et al. 2022, cap. 7) es un algoritmo tipo *dividir para vencer*; esto es, un algoritmo que divide un problema grande en instancias pequeñas más sencillas. Es uno de los algoritmos más veloces en la práctica por su buen manejo de memoria, aun cuando tiene un peor caso cuadrático, en promedio el costo es $O(n \log n)$.

① El arreglo se divide en 3 partes, ordenadas entre sí, un subarreglo izquierdo, un pivote, y un subarreglo derecho; los subarreglos no están ordenados localmente, pero el pivote está en su posición final.
② Se resuelve el problema izquierdo y el problema derecho por separado.
③ La función *part!* particiona el arreglo $A[low : end]$ en 3 partes como se especificó en el punto 1; para eso selecciona de manera aleatoria un pivote. Lo ponemos al final del arreglo para simplificar el código siguiente.
④ Este ciclo itera por todo el subarreglo, su objetivo es asegurar que $A[i] < piv$ para todo $i \in low : piv - 1$ y $piv < A[i]$ para todo $i \in piv + 1 : high$.
⑤ Intercambia elementos si $A[j] < piv$, hacemos seguimiento de i ya que esta posición determinará al pivote.
⑥ Como *piv* se encontraba en *high*, entonces hay que intercambiarlos para que *qsort!* sepa como manejarlos; recordando que los subarreglos no están ordenados dentro de sí.

```
8-element Vector{Int64}:
 1
 2
 3
 4
 5
```

```
6
7
8
```

El código Listado 4 es relativamente simple, usa recurrencias sobre *qsort!* sobre dos partes extremas divididas por un pivote; estos tres elementos son encontrados en *part!*. La función *part!* es muy eficiente en términos de memoria, lo que puede hacer la diferencia en la práctica. La correcta selección del pivote es muy importante para evitar casos malos, i.e., costo cuadrático; en esta implementación se realiza una selección aleatoria de pivote que funcionará en la mayoría de los casos.

El peor de los casos en *qsort!* es debido a una mala selección del pivote, de tal forma que

$$|A[low : piv - 1]| \ll |A[piv + 1 : high]|,$$

o lo contrario en toda selección, en el extremo uno de los subarreglos puede verse como de tamaño constante o cero, i.e., selección de pivote como el *mínimo* o el *máximo*. Esta estrategia reduce a *qsort!* a un costo $O(n^2)$.

Si se realiza un particionado donde

$$|A[low : piv - 1]| \approx |A[piv + 1 : high]|,$$

entonces tenemos un algoritmo $O(n \log n)$; ya que hace una división en dos partes casi iguales en cada recurrencia a *qsort!*, y esto solo puede profundizar $\log n$ veces, y en cada nivel *part!* tiene un costo lineal.

1.3. Skip list

Una *skip list* (Pugh 1990) es una lista ligada con capacidades de búsqueda eficiente con garantías probabilísticas, esto es que se cumplen con alta probabilidad. Para esto, la idea es que cada dato tiene asociado un arreglo de punteros o referencias hacia nodos sucesores, i.e., los nodos a nivel i se conectan con el siguiente nodo a nivel i. En el nivel más bajo, la *skip list* es una simple lista ligada, mientras que sube, se vuelve más dispersa dando saltos más largos.

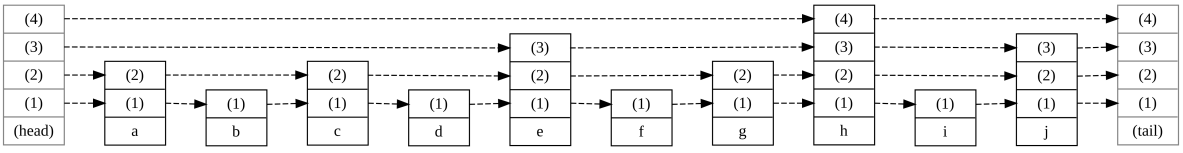

Figura 1: Ejemplo de una skip list

A diferencia de los algoritmos vistos anteriormente, en este caso, ya se tiene una estructura de datos, que conlleva un costo en memoria explícito por nodo. Figura 1 ilustra la estructura.

La altura de cada nodo es calculada de manera probabilística, dada la probabilidad p. Un valor común de $p = 0.5$. La altura de cada nodo se calcula como sigue:

```
function levels(p)
  i = 1
  while rand() < p
    i += 1
  end

  i
end
```

```
levels (generic function with 1 method)
```

Si tenemos n evaluaciones de *levels*, los niveles pequeños son relativamente probables, mientras que niveles grandes son relativamente poco probables. De hecho, los niveles $\log_{1/p} n$ son cercanos a una constante, $\log_{1/p} n - 1$ son $1/p$ veces la constante, $\log_{1/p} n - 2$ son $1/p^2$ veces la constante, etc.

A diferencia de los algoritmos anteriores, una *skip list* comienza vacía, y se va poblando insertando elementos a la lista. Se va colocando en la posición que no viole el orden; generando el nodo correspondiente con nivel calculado. Los nodos especiales *head* y *tail* siempre tienen el nivel máximo posible. La inserción de un valor encapsulado en el nodo u comienza por visitar el máximo nivel en *head* e ir bajando hasta determinar $u.dato > head[level].dato$; en ese momento se debe avanzar al nodo apuntado por $head[level]$ y repetir el algoritmo hasta que $level = 1$, en cuyo caso encontramos el lugar de inserción del nuevo dato. Se procede a reasignar los punteros de los sucesores y ajustar los punteros hacia los nodos sucesores a los niveles que tiene u.

Cada inserción tiene un costo $O(\log_{1/p} n)$, garantía probabilística; por lo que insertar n elementos tiene un costo:

$$\sum_{i=1}^{n} O(\log_{1/p} i) = O(\log_{1/p} \prod_{i=1}^{n} i) = O(\log_{1/p} n!) = O(n \log n);$$

usando la aproximación de Stirling.

A diferencia de la versión basada en arreglos, una *skip list* es capaz de aceptar nuevos elementos y mantener el orden de manera eficiente.

1.3.1. Ejercicios:

1. Investigue, implemente y pruebe *merge sort*. 1.1 ¿Cuáles son las ventajas y desventajas de *merge sort*? 1.2 ¿Por qué *merge sort* se puede utilizar en algoritmos paralelos y otros pueden tener muchas dificultades? 1.3 ¿Cómo se puede reducir la memoria extra necesaria de *merge sort*?
2. Investigue, implemente y pruebe *heap sort*. 2.1 ¿Cuáles son las ventajas y desventajas de *heap sort*?
3. ¿Cuál es el costo en memoria de una *skip list*? 3.1 Investigue, implemente y pruebe una *skip list*.
4. Investigue, implemente y pruebe un árbol binario de búsqueda.

1.4. Lecturas

Las lecturas de este tema corresponden al capítulo 5 de (Knuth 1998), en específico 5.2 *Internal sorting*. También se recomienda leer y comprender la parte II de (Cormen et al. 2022), que corresponde a *Sorting and order statistics*, en particular Cap. 6 y 7, así como el Cap. 8.1. El artículo de Wikipedia https://en.wikipedia.org/wiki/Sorting_algorithm también puede ser consultado con la idea de encontrar una explicación rápida de los algoritmos.

En la práctica, pocos algoritmos son mejores que *quicksort*. En (Loeser 1974) se detalla una serie de experimentos donde se compara quicksort contra otros algoritmos relacionados; por lo que es una lectura recomendable.

La parte adaptable, esto es para algoritmos *oportunistas* que toman ventaja de instancias simples, está cubierta por el artículo (Estivill-Castro y Wood 1992). En especial, es muy necesario comprender las secciones 1.1 y 1.2, el resto del artículo debe ser leído aunque no invierta mucho tiempo en comprender las pruebas expuestas si no le son claras. En especial, en las secciones indicadas se establecen las medidas de desorden contra las cuales se mide la complejidad. En (Cook y Kim 1980) se realiza una comparación del desempeño de varios algoritmos para ordenamiento de listas casi ordenadas, esto es, en cierto sentido donde los algoritmos adaptables tienen sentido. Este artículo es anterior a (Estivill-Castro y Wood 1992) pero tiene experimentos que simplifican el entendimiento de los temas.

1.5. Material audio-visual sobre algoritmos de ordenamiento

1.6. Actividades

1. Implementa y compara los siguientes algoritmos de ordenamiento:

   - heapsort
   - mergesort
   - quicksort
   - bubblesort adaptativo

2. Utiliza los diferentes archivos proporcionados, los cuales tienen diferentes niveles de desorden y mide tanto el número de comparaciones como el tiempo necesario para ordenarlos.
3. Por cada archivo de datos, compara todos los métodos implementados mediante figuras o tablas de datos (número de comparaciones y tiempo por separado).
4. Discute tus resultados.

1.6.1. Entregable

Su trabajo se entregará en PDF y con el notebook fuente, se entregará en el sistema en archivos separados. El código se debe documentar, así como mantener una estructura del documento que permita a un lector interesado entender el problema, sus experimentos y metodología, así como sus conclusiones. Tenga en cuenta que los notebooks pueden alternar celdas de texto y código.

No olvide estructurar su reporte, en particular el reporte debe cubrir los siguientes puntos:

- Título del reporte, su nombre.
- Introducción.
- Desarrollo de las actividades.
- Conclusiones
- Lista de referencias. Nota, una lista de referencias que no fueron utilizadas en el cuerpo del texto será interpretada como una lista vacía.

Nota: no olvides revisar la rúbrica del curso.

Cook, Curtis R, y Do Jin Kim. 1980. "Best sorting algorithm for nearly sorted lists". *Communications of the ACM* 23 (11): 620–24.

Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest, y Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

Estivill-Castro, Vladmir, y Derick Wood. 1992. "A survey of adaptive sorting algorithms". *ACM Computing Surveys (CSUR)* 24 (4): 441–76.

Knuth, Donald. 1998. *The Art Of Computer Programming, vol. 3 (2nd ed): Sorting And Searching*. Vol. 3. Addison Wesley Longman Publishing Co. Inc.

Loeser, Rudolf. 1974. "Some performance tests of 'quicksort' and descendants". *Communications of the ACM* 17 (3): 143–52.

Pugh, William. 1990. "Skip lists: a probabilistic alternative to balanced trees". *Commun. ACM* (New York, NY, USA) 33 (6): 668–76. https://doi.org/10.1145/78973.78977.

Listado 4 Algoritmo *quick sort*.

```
using Random

function qsort!(A, low=1, high=length(A))
  if low < high
      piv = part!(A, low, high)                                    ①
      qsort!(A, low, piv - 1)                                      ②
      qsort!(A, piv + 1, high)
  end

  A
end

function part!(A, low, high)
  ipiv = rand(low:high)                                            ③
  A[ipiv], A[high] = A[high], A[ipiv]
  piv = A[high]

  i = low - 1  # uno antes porque se accede después de un i+1
  for j in low:high - 1                                            ④
      if A[j] < piv
          i += 1
          A[i], A[j] = A[j], A[i]                                  ⑤
      end
  end

  ipiv = i + 1
  A[ipiv], A[high] = A[high], A[ipiv]                              ⑥
  ipiv
end

qsort!([6, 8, 3, 7, 4, 1, 2, 5])
```