Rappels(?) d'algorithmique

Chapter content:

- Tris
- Data structures
- Compression

Table des matières

Comparison sort (quizz)

Where is the snag (rather obvious)?

Algorithm 1: Sort algo in $O(n \log n)$

Input: $S = \{i_1, \ldots, i_n\}$ unsorted.
Output: $A = [o_1, \ldots, o_n]$ contains S, sorted.

$A = [\,]$
while $S \neq \emptyset$
  Remove one item x from S // no matter which: O(1)
  Insert x into A at proper position // binary search in $O(\log n)$
end
return A

Claim: each loop in $O(\log n)$ so $O(n \log n)$ overall (?)

Performance of sorts (1): comparison sorts

https://en.wikipedia.org/wiki/Sorting_algorithm

3 main algorithms to sort efficiently ($n \log n$):

- MergeSort
- HeapSort
- QuickSort *(worst case $O(n^2)$, but fast on average)*

Hybrids: IntroSort (QuickSort + HeapSort), TimSort (InsertionSort + MergeSort)

Ex: gnu c++ library;

- *Introsort*: tries *quicksort*, then, if recursion exceed depth $2 \log n$, switches to heapsort.
- when input size drops below <16, insertion sort

Ex: *TimSort* (Python, Android, Java SE7, Octave);

- searches runs of consecutive items, combines small runs through insertion sort.
- applies MergeSort on sequences obtained. The merging algorithm is not trivial.

Performance of sorts (2): sorting integers

https://en.wikipedia.org/wiki/Sorting_algorithm

- Input: a list of objects $o_1, \ldots, o_n$. Each has integer weight $1 \leq w_i \leq n$. How would you sort?
- And if the weight is an integer between n et n^5?

Beware: $\Omega(n \log n)$ only holds for comparison sort.

But do not expect better in practice. Even for integers.

Sorting Algorithms

https://en.wikipedia.org/wiki/Sorting_algorithm

Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$ variation is n	$n \log n$	n^2	$\log n$ on average, worst case space complexity is n; Sedgewick variation is $\log n$ worst case.	Typical in-place sort is not stable; stable versions exist.	Partitioning	Quicksort is usually done in-place with O(log *n*) stack space.[4][5]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n A hybrid block merge sort is O(1) mem.	Yes	Merging	Highly parallelizable (up to O(log *n*) using the Three Hungarians' Algorithm[6] or, more practically, Cole's parallel merge sort) for processing large amounts of data.
In-place merge sort	—	—	$n \log^2 n$ See above, for hybrid, that is $n \log n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[7]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	O(*n* + *d*), in the worst case over sequences that have *d* inversions.
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, for example using lists.[8]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes *n* comparisons when the data is already sorted or reverse sorted.
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes *n* comparisons when the data is already sorted or reverse sorted.
Shell sort	$n \log n$	Depends on gap sequence	Depends on gap sequence; best known is $n^{4/3}$	1	No	Insertion	Small code size, no use of call stack, reasonably fast, useful where memory is at a premium such as embedded and older mainframe applications. There is a worst case O(n (log n)²) gap sequence but it loses O(n log n) best case time.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Binary tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes.
Library sort	n	$n \log n$	n^2	n	Yes	Insertion	
Patience sorting	n	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in O(*n* log *n*).
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	$n \log n$	$n \log n$	$n \log n$	n[9]	No	Selection	Variation of Heap Sort.
Cocktail sort	n	n^2	n^2	1	Yes	Exchanging	
Comb sort	$n \log n$	n^2	n^2	1	No	Exchanging	Faster than bubble sort on average.
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
UnShuffle Sort[10]	n	kn	kn	In-place for linked lists. n × sizeof(link) for array.	No	Distribution and Merge	No exchanges are performed. The parameter *k* is proportional to the entropy in the input. *k = 1* for ordered or reverse ordered input.
Franceschini's method[11]	—	$n \log n$	$n \log n$	1	Yes	?	
Block sort	n	$n \log n$	$n \log n$	1	Yes	Insertion & Merging	Combine a block-based O(n) in-place merge algorithm[12] with a bottom-up merge sort.
Odd–even sort	n	n^2	n^2	1	Yes	Exchanging	Can be run on parallel processors easily.

Table des matières

Data structures

https://en.wikipedia.org/wiki/Sorting_algorithm

Dictionnary=associative array: manage key-value pairs.
(Worst-case) complexity guarantee:

hashmap

search	insert	delete
$O(1)$	$O(1)$	$O(1)$

on average

Maintain a set "ordered" for min/max queries?

heap (Fibonacci...)

find_min	delete_min	insert	merge
$O(1)$	$O(\log n)$	$O(1)$	$O(1)$

If we want to support predecessor/succesor/ i^{th}?

tree (AVL, RB)

search	insert	delete	pred	$i^{\text{ème}}$
$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

or skip list, but then *on average*

Bloom Filter

Bloom filter:

structure de données probabiliste pour tester l'appartenance à un ensemble:

- 1 vecteur A de m bits
- k fonctions de hachage $h_1, \ldots, h_k$

Vecteur A pour l'ensemble S:

Au départ $A = [0, \ldots 0]$. Puis
$\forall x \in S \, \forall i \leq k$, set $A[h_i(x)]$ to 1.

To test if $y \in S$:

if $(\bigwedge_i A[h_i(y)]) = 0$: $y \notin S$
if $(\bigwedge_i A[h_i(y)]) = 1$: claim $y \in S$ (probably true)

Proba. of false positive:
$p \approx (1 - e^{-kn/m})^k$.

Usage:

- partition pruning (ex: joins, filters in Oracle)
- semijoins (DB2 star query optimization)

Table des matières

Compression: (reminders)

A dataset can be compressed efficiently if it has low entropy (= is redundant)

Huffman algorithm (or arithmetic encoding) yields optimal prefix code, but
- complex (need frequencies), often too slow
- encodes each item separately (item=symbol?)

DB applications: compress vectors. For instance, compress columns of a table (in column store). Especially useful if table is sorted on the column.

Vector compression: widespread algorithms

- Run-length Encoding:
  Ex: [a,a,a,b,b,d,e,e,e,e,e,e,a,a]
  Compressed Vector: (a,3) (b,2) (d,1) (e,6) (a,2)

  Remark: some versions record the starting offset for the run instead of length (or both).

- Delta Encoding:
  Ex: [22100,22100,22101,22101,22070,22105,22105]
  Compressed Vector: [22100,0,1,0,-31,35,0]

- Offset Encoding
  Ex: [12,0,0,0,105,0,0,0,0,104]
  Compressed Vector: (0,12) (4,105) (9,104)

- ...

Often, vector is split in blocks, which are compressed independently.

Challenge: perform operations directly on compressed data.

Illustration: JPEG compression

[https://en.wikipedia.org/wiki/JPEG]

Conversion + sampling → Split in blocks of 8x8 pixels → DCT → Quantization → RLE+Huffman

Compression

after blocks

$$\begin{bmatrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{bmatrix}$$

after quantization

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

after conversion

3 components

Y' C_B C_R

after DCT

$$\begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.12 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.87 & 1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & -5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.68 \end{bmatrix}$$

after RLE

(0, 2)(-3);
(1, 2)(-3);
(0, 1)(-2);
(0, 2)(-6);
...

run length
#bits
value

The Snappy library

[https://github.com/google/snappy]

Objective: maximize (de)compression speed, rather than compression rate.
Devised within Google, used in many modern DBMS.

General Idea: compressed version first specifies the length of original text (as a varint), then 2 element types (same idea as for LZ77 compression)

- elements copied as is from original text
- elements recording offset and length from a repeated prefix