Sorting

Hsuan-Tien Lin

Dept. of CSIE, NTU

April 13, 2021

Selection Sort: Review and Refinements

idea: linearly select the minimum one from "unsorted" part;
put the minimum one to the end of the "sorted" part

Implementations

- common implementation: swap minimum with $a[i]$ for putting in i-th iteration

  a[10]

- rotate implementation: rotate minimum down to $a[i]$ in i-th iteration
- linked-list implementation: insert minimum to the i-th element

- space $O(1)$: in-place
- time $O(n^2)$ and $\Theta(n^2)$
- rotate/linked-list: stable by selecting minimum with smallest index —same-valued elements keep their index orders
- common implementation: unstable

Heap Sort: Review and Refinements

idea: selection sort with a max-heap in original array
rather than unordered pile

- space $O(1)$
- time $O(n \log n)$
- not stable
- usually preferred over selection (faster)

Insertion Sort: Review and Refinements

idea: insert a card from the unsorted pile to its place in the sorted pile

Implementations

- naive implementation: sequential search sorted pile from the front
  $O(n)$ time per search, $O(n)$ per insert
- backwise implementation: sequential search sorted pile from the back $O(n)$ time per search, $O(n)$ per insert
  stable
- binary-search implementation: binary search the sorted pile
  $O(\log n)$ time per search, $O(n)$ per insert
- linked-list implementation: same as naive but on linked lists
  $O(n)$ time per search, $O(1)$ per insert
- rotation implementation: neighbor swap rather than insert
  (gnome sort)

Insertion Sort: Review and Refinements (II)

- space $O(1)$
- time $O(n^2)$
- stable
- backwise implementation adaptive
- usually preferred over selection (adaptive)

Shell Sort: Introduction

idea: adaptive insertion sort on every k_1 elements;
adaptive insertion sort on every k_2 elements; $\cdots$
adaptive insertion sort on every $k_m = 1$ element

- insertion sort with "long jumps"
- space $O(1)$, like insertion sort
- time: difficult to analyze, often faster than $O(n^2)$
- unstable, adaptive
- usually good practical performance and somewhat easy to implement

Merge Sort: Introduction

idea: combine sorted parts repeatedly to get everything sorted

Implementations

- bottom-up implementation:

6	5	4	7	8	3	1	2	(size-1 sorted)
5	6	4	7	3	8	1	2	(size-2 sorted)
4	5	6	7	1	2	3	8	(size-4 sorted)
1	2	3	4	5	6	7	8	(size-8 sorted)

log n

n

  - $O(\log n)$ loops, the i-th loop combines size-2^i arrays $O(n/2^i)$ times
  - combine size-ℓ array can take $O(\ell)$ time but need $O(\ell)$ space! (how about lists?)
  - thus, bottom-up Merge Sort takes $O(n \log n)$ time
- top-down implementation:

nlogn

MergeSort(arr, left, right)

= combine(MergeSort(arr, left, mid), MergeSort(arr, mid+1, right));

  - divide and conquer, $O(\log n)$ level recursive calls

Merge Sort: Review and Refinements

idea: combine sorted parts repeatedly to get everything sorted

- time $O(n \log n)$ in both implementations
- usually stable (if carefully implemented), parallellize well
- popular in external sort

Tree Sort: Review and Refinements

idea: replace heap with a BST;
an in-order traveral outputs the sorted result

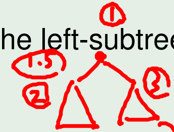
- space $O(n)$
- time: $O(n \cdot h)$, with worst $O(n^2)$ (unbalanced tree), average $O(n \log n)$, careful BST $O(n \log n)$
- unstable
- suitable for stream data and incremental sorting

Quick Sort: Introduction

idea: simulate tree sort without building the tree

Tree Sort Revisited

make $a[0]$ the root of a BST
for $i \leftarrow 1, \cdots, n-1$ **do**
  if $a[i] < a[0]$
    insert $a[i]$ to the left-subtree of BST
  else
    insert $a[i]$ to the right-subtree of BST
  end if
end for
in-order traversal of left-subtree, then root, then right-subtree



Quick Sort

name $\boxed{a[0]}$ the pivot
for $i \leftarrow 1, \cdots, n-1$ **do**
  if $a[i] < a[0]$
    put $a[i]$ to the *left* pile of the pivot
  else
    put $a[i]$ to the *right* pile of the pivot
  end if
end for
output quick-sorted *left*; output $a[0]$; output quick-sorted *right*

Quick Sort Simulation

$\boxed{6}$, 1, 4, 9, 7, 8, 3, 10, 2, 5

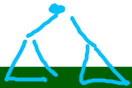
→ 1 4 5 2 3 8 10 7 9 ← $\boxed{6}$

$\boxed{3}$ 1 4 5 2 $\boxed{6}$ 8 10 7 9

1, 4, 3, 2, 5

2 1 ~~2~~ 5 4

$\boxed{3}$



pivot

Implementations

- naive implementation: pick first element in the pile as pivot
- random implementation: pick a random element in the pile as pivot
- median-of-3 implementation: pick median(front, middle, back) as pivot

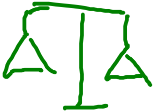
- space: worst $O(n)$, average $O(\log n)$ on stack calls
- time: worst $O(n^2)$, average $O(n \log n)$
- not stable

long jump

- usually best choice for large data (if not requiring stability), can be mixed with other sorts for small data

9 coins

two uses



A B C D E F G H I

one of them heavier

ABC ? DEF

ABC > | = GHI | < DEF

A ? B | GHI | D ? E

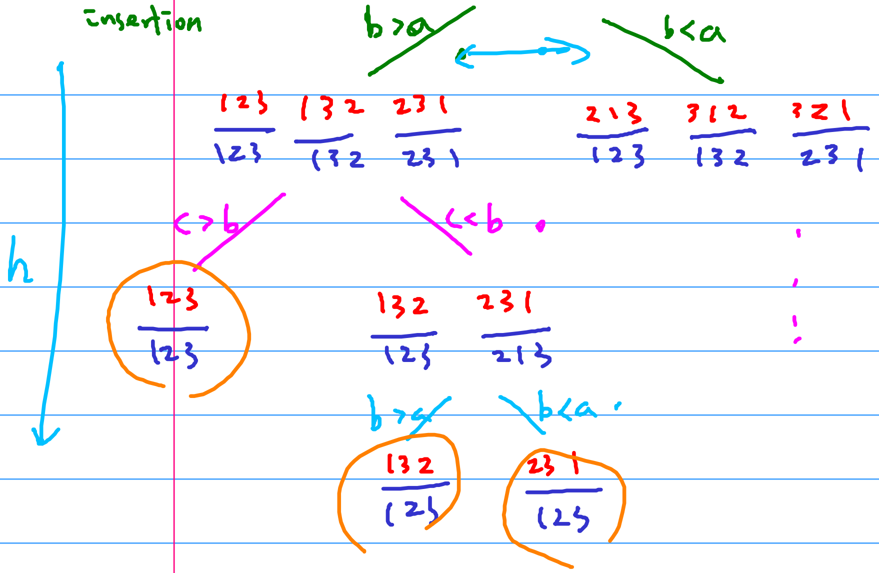
\> A | = C | < B | G | I | H | \> D | = F | < E

trinary tree

≥ 9 leaf

abc : 123, 132, 213, 231, 312, 321

insertion



height h, binary tree

leaf $\leq 2^h$

$\#$ leaf $\geq$ $\#$ permutation

$n!$

$\Rightarrow$ min $\#$ comparison

$\Omega(\log n!)$

$\Omega(n \log n)$

$$C_{avg}(0) = 0$$

$$\left(C_{avg}(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}\left[C_{avg}(i-1) + C_{avg}(n-i) \right] \right.$$

$$\left. = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} C_{avg}(i) \right) \cdot n$$

$$- \left(C_{avg}(n-1) = (n-2) + \frac{2}{n-1}\sum_{i=0}^{n-2} C_{avg}(i) \right) \cdot (n-1)$$

$$\Rightarrow \quad n \cdot C(n) - (n-1)\, C(n-1) = 2(n-1) + 2\, C(n-1)$$

$$n \cdot C(n) = 2(n-1) + (n+1)\, C(n-1)$$

$$\frac{C(n)}{n+1} = \cancel{\frac{C(0)}{1}} + \sum_{i=1}^{n} \frac{2}{i} - \cancel{\sum_{i=1}^{n} \frac{4}{i(i+1)}}$$

$O(\log n)$

$C_{avg} = O(n \log n)$