sorting

used in many different settings

also artistic impressions, even dance

**puzzle:** can we sort 5 elements in 7 comparisons?
overview

- heap sort: more
- heapsort
- priority queues
- quicksort
- material
overview

- heap sort: more
- heapsort
- priority queues
- quicksort
- material
crucial procedures for heapsort

maxHeapify (or downMaxHeap)

buildMaxHeap

heapSort
Algorithm maxHeapify($A, i$):

1. $l := \text{left}(i)$
2. $r := \text{right}(i)$
3. if $l \leq A.\text{heap-size}$ and $A[l] > A[i]$ then
   - $\text{largest} := l$
4. else
   - $\text{largest} := i$
5. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$ then
   - $\text{largest} := r$
6. if $\text{largest} \neq i$ then
   - swap($A[i], A[\text{largest}]$)
   - maxHeapify($A, \text{largest}$)
maxHeapify: correctness

induction on the height of node i
if the height is 0, then immediate

if the height is > 0, then two cases

case largest = i immediate

case largest = l (and equivalent for largest = r) use induction
maxHeapify: time complexity

with $h$ the height of the heap:

$$T(h) = T(h - 1) + 1 \text{ if } h > 0 \text{ gives } T(h) \in \mathcal{O}(h)$$

then use $h \in \Theta(\log n)$

with $n$ the number of nodes of the heap:

$$T(n) = T(\frac{2}{3}n) + 1 \text{ if } n > 1 \text{ gives } T(n) \in \mathcal{O}(\log n)$$

because in the worst case the bottom level is exactly half full
Algorithm buildMaxHeap(\(H\)):

\[
H.size := H.length
\]

for \(i = \lfloor H.length/2 \rfloor\) downto 1 do

donMaxHeap(H, i)
buildMaxHeap: correctness

use loop invariant:

at the start of the for-loop each node \( i + 1, \ldots, n \) is the root of a max-heap

init:
for \( i = \lceil \frac{n}{2} \rceil \) the nodes \( i + 1, \ldots, n \) are leaves

loop:
children are max-heaps

use correctness of maxHeapify

end:
for \( i = 0 \) the invariant gives correctness of the output
buildMaxHeap: complexity

rough estimation:
for each of the \( \lfloor \frac{n}{2} \rfloor \) internal nodes of the heap
we do maxHeapify which is in \( O(h) \)
so buildMaxHeap in \( O(n \cdot \log n) \)

more precise estimation:
for every height \( j \) in 0, \ldots, \lfloor \log n \rfloor
for each of the at most \( \lceil \frac{n}{2^{j+1}} \rceil \) nodes of height \( j \)
we do downMaxHeap which is in \( O(j) \)
so (!) buildMaxHeap is in \( O(n) \)
buildMaxHeap: alternative algorithm?

from 1 to ⌊\frac{n}{2}⌋?

ty = 1, 3, 2, 4, 5, 6, 7] with i = 1, 2, 3
Algorithm heapsort($H$):
   buildMaxHeap($H$)
   for $i = H$.length downto 2 do
      swap $H[1]$ and $H[i]$
      $H$.heap-size := $H$.heap-size − 1
      downMaxHeap($H$, 1)
heapSort: analysis

running time in $O(n \log n)$

what happens to a sorted input?
what is the best-case?
Smoothsort, an alternative for sorting in situ

by Edsger W. Dijkstra,
Burroughs Corporation

Abstract.

Like heapsort—which inspired it—smoothsort is an algorithm for sorting in situ. It is of order $N \cdot \log N$ in the worst case, but of order $N$ in the best case, with a smooth transition between the two. (Hence its name.)

Key Words and Phrases: sorting in situ, heapsort, sorting trees, sift, computational complexity.

CR Categories: 5.25, 5.31
smooth sort: inventor

Edsger W. Dijkstra 1930–2002

Turing Award 1972
overview

- heap sort: more
- heapsort
- priority queues
- quicksort
- material
priority queue

data type for maintaining a set

access via keys

queue where most important element is served first

most important: minimum key or maximum key
priority queue: abstract data type

\[\text{insert}(Q, x)\] inserts element \(x\) in the (max- or min-)priority queue \(Q\)

for max-priority queue:

\[\text{extractMax}(Q)\] removes element with maximum key

\[\text{maximum}(Q)\] returns (but does not remove) element with max key
priority queue: implementation

use a heap

max-heap for max-priority queue

min-heap for min-priority queue
Algorithm maximum($H$):
    return $H[1]$
Algorithm extractMax(H):

\[
\begin{align*}
\text{max} & \ := \ H[1] \\
H[1] & \ := \ H[H.\text{heap-size}] \\
H.\text{heap-size} & \ := \ H.\text{heap-size} - 1 \\
& \text{maxHeapify}(H, 1) \\
& \text{return} \ \text{max}
\end{align*}
\]

running time?
remove: pseudo-code

$H$ max-heap; remove and return maximum; error omitted

**Algorithm** extractMax($H$):

```plaintext
max := H[1]
H.heap-size := H.heap-size - 1
maxHeapify(H, 1)
return max
```

running time? in $O(\log n)$
insert for max-priority queue: pseudo-code

$H$ a max-heap; insert key $k$; bubble upwards

**Algorithm** insert($H, k$):

$H$.heap-size := $H$.heap-size + 1

$H[H$.heap-size$] := -\infty$

HeapIncreaseKey($H, H$.heap-size, $k$)

**Algorithm** HeapIncreaseKey($H, i, k$):

if $k < H[i]$ then
    return error

$H[i] := k$

while $i > 0$ and $H[parent(i)] < H[i]$ do
    swap($H[parent(i)]$, $H[i]$)
    $i := parent(i)$
example: insert 

add key 19 to the max-heap [15, 13, 9, 5, 12, 8, 7, 4, 1, 6]
priority queue operations: running time

- insert in $\mathcal{O}(\log n)$
- remove in $\mathcal{O}(\log n)$
- maximum or minimum in $\mathcal{O}(1)$
questions about algorithms

give the intuition of the algorithm
give the pseudo-code for the algorithm
apply the algorithm
adapt the algorithm
analyse correctness of the algorithm
analyse (worst-case) time complexity of the algorithm
overview

- heap sort: more
- heapsort
- priority queues
- quicksort
- material
quicksort

merge sort is in $\mathcal{O}(n \log n)$ but it is not in-place

quicksort is also a divide-and-conquer sorting algorithm

it can partly solve the drawback of merge sort
quicksort: idea

**divide:** partition the input-array $A$ into two parts:
up to index $q$ containing keys less than or equal to pivot
starting index $q + 1$ containing keys larger than pivot

**recur:** sort recursively the two sub-arrays

**conquer:** combine the two partial solutions into one sorted array
here there is nothing to do
quicksort: pseudo-code

we sort the (sub-)array $A[p \ldots r]$

**Algorithm** quickSort($A, p, r$):

*if* $p < r$ *then*

$q := \text{partition}(A, p, r)$

quickSort($A, p, q - 1$)

quickSort($A, q + 1, r$)
quicksort: pseudo-code

we sort the (sub-)array \( A[p \ldots r] \)

**Algorithm** `quickSort(A, p, r)`:  
if \( p < r \) then  
\( q := \text{partition}(A, p, r) \)  
`quickSort(A, p, q - 1)`  
`quickSort(A, q + 1, r)`

what happens if \( p = r \)?
while the list contains more than one element:

take a number $k$ from the list, called the pivot

index $i$ indicates the last key of the small-ones-so-far

index $j$ indicates the first key to be compared with the pivot

if $j$ finds a key smaller than pivot
swap that key with the one at $i + 1$
Algorithm partition(A, p, r):
    x := A[r]
    i := p − 1
    for j = p to r − 1 do
        if A[j] ≤ x then
            i := i + 1
            exchange A[i] with A[j]
        exchange A[i + 1] with A[r]
    return i + 1
Algorithm partition($A, p, r$):

$x := A[r]$

$i := p - 1$

for $j = p$ to $r - 1$ do

if $A[j] \leq x$ then

$i := i + 1$

exchange $A[i]$ with $A[j]$

exchange $A[i + 1]$ with $A[r]$

return $i + 1$

quickSort calls partition only if $p < r$
**Algorithm** partition\( (A, p, r) \):

\[
x := A[r] \\
i := p - 1 \\
\text{for } j = p \text{ to } r - 1 \text{ do} \\
\quad \text{if } A[j] \leq x \text{ then} \\
\quad \quad i := i + 1 \\
\quad \quad \text{exchange } A[i] \text{ with } A[j] \\
\quad \text{exchange } A[i + 1] \text{ with } A[r] \\
\text{return } i + 1
\]

quickSort calls partition only if \( p < r \)

running time?
Algorithm partition(A, p, r):

\[
x := A[r] \\
i := p - 1 \\
\text{for } j = p \text{ to } r - 1 \text{ do} \\
\quad \text{if } A[j] \leq x \text{ then} \\
\quad \quad i := i + 1 \\
\quad \quad \text{exchange } A[i] \text{ with } A[j] \\
\quad \text{exchange } A[i + 1] \text{ with } A[r] \\
\text{return } i + 1
\]

quickSort calls partition only if \( p < r \)

running time? \( r - p \) steps so in \( \mathcal{O}(r - p + i) = \mathcal{O}(n) \)
question: apply partition

3 6 1 7 8 2 5 4
question: give recursion tree for quicksort

3 6 1 7 8 2 5 4
**INEFFECTIVE SORTS**

**DEFINE HALFHEARTEDMERCESORT(list):**

IF LENGTH(list) < 2:
    RETURN list

Pivot = INT(LENGTH(list) / 2)
A = HALFHEARTEDMERCESORT(list[:Pivot])
B = HALFHEARTEDMERCESORT(list[Pivot:])

// UMMM
RETURN [A, B] // HERE.. SORRY.

**DEFINE FASTBOGOSORT(list):**

// AN OPTIMIZED BOGOSORT
// RUNS IN O(N LOG N)

FOR n FROM 1 TO LOG(LENGTH(list)):
    SHUFFLE(list):
    IF ISORTED(list):
        RETURN list

RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"

**DEFINE JIBBERJABBERQUICKSORT(list):**

OK SO YOU CHOOSE A PIVOT
THEN DIVIDE THE LIST IN HALF
FOR EACH HALF:
    CHECK TO SEE IF IT'S SORTED
    NO, WHAT DOESN'T MATTER
    COMPARE EACH ELEMENT TO THE PIVOT
    THE BIGGER ONES GO IN A NEW LIST
    THE SMALLER ONES GO INTO, UH
    THE SECOND LIST FROM BEFORE

HANG ON, LET ME NAME THE LISTS
THIS IS LIST A
THE NEW ONE IS LIST B
PUT THE BIG ONES INTO LIST B
NOW TAKE THE SECOND LIST
CALL IT LIST UH, A2
WHICH ONE WAS THE PIVOT IN?
SCRATCH ALL THAT
IT JUST RECURSIVELY CALLS ITSELF
UNTIL BOTH LISTS ARE EMPTY
RIGHT?

NOT EMPTY, BUT YOU KNOW WHAT I MEAN
AM I ALLOWED TO USE THE STANDARD LIBRARIES?

**DEFINE PANICSORT(list):**

IF ISORTED(list):
    RETURN list

FOR n FROM 1 TO 1000:
    Pivot = RANDOM(0, LENGTH(list))
    List = List[Pivot:] + List[:Pivot]
    IF ISORTED(list):
        RETURN list

IF ISORTED(list):
    RETURN list

IF ISORTED(list):
    // THIS CAN'T BE HAPPENING
    RETURN list

IF ISORTED(list):
    // COME ON COME ON
    RETURN list

// OH JEEZ
// I'M GONNA BE IN SO MUCH TROUBLE
LIST = []
SYSTEM("SHUTDOWN -H +5")
SYSTEM("RM -RF .")
SYSTEM("RM -RF ~/")
SYSTEM("RM -RF /")
SYSTEM("RD /5 & Q.CV") // PORTABILITY
RETURN [1, 2, 3, 4, 5]
partition: correctness

property: at the beginning of the for-loop

- if \( p \leq k \leq i \) then \( A[k] \leq pivot \)
  the part up to (included) \( i \) contains the small-ones-so-far

- if \( i + 1 \leq k \leq j - 1 \) then \( A[k] > pivot \)
  the ones not selected for swapping by \( j \) are large-ones-so-far

- if \( k = r \) then \( A[k] = pivot \)
  pivot does not move during iterations of for-loop
partition: correctness

does the property hold initially?
is the property maintained during the loop?
if so, then it is an invariant! does it yield correctness?
quicksort: correctness

follows from correctness of partition and induction
overview

- heap sort: more
- heapsort
- priority queues
- quicksort
- material
extra material

- smooth sort
- sorting wikipedia