data structures and algorithms
2016 09 13
lecture 3

done

sorting algorithms
insertion sort (book 1, 2)
selection sort (book 1, 2)
bubble sort (book 1, 2)
merge sort (book 2)

paradigm and techniques
divide–and–conquer, recursion tree, recurrence equation (book 2, 4.3, 4.4)

asymptotic complexity
intuitive (and book 3.1, today)

overview

• recap
• asymptotic bounds
• trees and binary trees
• heaps
• heap bubble
• building a heap
• heapsort

overview

• recap
• asymptotic bounds
• trees and binary trees
• heaps
• heap bubble
• building a heap
• heapsort
asymptotic bounds: intuition

what is the behaviour of the function for large inputs

Θ asymptotic tight bound (‘=’)

Ο asymptotic upper bound (‘≤’)

Ω asymptotic lower bound (‘≥’)

o asymptotic upper bound not tight (‘<’)

ω asymptotic lower bound not tight (‘>’)

asymptotic upper bound: big-Ο

for f, g : N → R⁺: f(n) ∈ O(g(n)) if

∃c, c' > 0 ∈ R
∃n₀ > 0 ∈ N
∀n ∈ N : n ≥ n₀ ⇒ f(n) ≤ c · g(n)

f(n) is eventually ‘as large as g(n)’ up to a constant

Θ in terms of big-Ο

alternative definition of Θ using O:

for f, g : N → R⁺: f(n) ∈ Θ(g(n)) if

f(n) ∈ O(g(n)) and

f(n) ∈ O(f(n))

f(n) is eventually smaller-equal g(n) up to constants
big-\text{O}: simple examples

\[ 3n^2 + 5n + 8 \in O(n^2) \]
\[ 3n^2 + 5n + 8 \in \Theta(n^2) \]
\[ 3n^2 + 5n + 8 \in O(n^3) \]
\[ 3n^3 + 5n + 8 \notin \Theta(n^3) \]
\[ 1000 \in \Theta(1) \]

\[ 3n^2 + 5n + 8 \in O(n^2) \]
\[ 3n^2 + 5n + 8 \in \Theta(n^2) \]
\[ 3n^2 + 5n + 8 \in O(n^3) \]
\[ 3n^3 + 5n + 8 \notin \Theta(n^3) \]
\[ 1000 \in \Theta(1) \]

big-\text{O}: some properties

\[ n^a \in O(n^b) \text{ for all } 0 < a \leq b \quad \frac{1}{2}n^2 \in O(n^3) \]
\[ \log_a(n) \in O(n^b) \text{ for all } a, b > 0 \quad \log n \in O(\sqrt{n}) \]
\[ n^a \in O(b^n) \text{ for all } a > 0 \text{ and } b > 1 \quad n^5 \in O(2^n) \]
\[ \log_a n \in O(\log_b n) \text{ for all } a, b > 0 \quad \log_2 n \in O(\log_3 n) \]

the latter because

\[ a^\log_a b^{\log_b n} = (a^{\log_a b})^{\log_b n} \]
\[ = b^{\log_b n} \]
\[ = n \]

hence we have:
\[ \log_a b \cdot \log_b n = \log_a n \]

\( \Theta \) and big-\text{O}

worst-case time complexity of insertion sort is in \( O(n^2) \)

time complexity of insertion sort is in \( O(n^2) \)

worst-case time complexity of insertion sort is in \( \Theta(n^2) \)

time complexity of insertion sort is not in \( \Theta(n^2) \) (sometimes better!)

growth of some important functions
growth of some functions: table

<table>
<thead>
<tr>
<th>n</th>
<th>log n</th>
<th>n log n</th>
<th>n^2</th>
<th>n^3</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>62</td>
<td>512</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>32</td>
<td>10^3</td>
<td>10^3</td>
<td>13000</td>
<td>10^6</td>
<td>4294967296</td>
</tr>
<tr>
<td>10^3</td>
<td>10^3</td>
<td>10^3</td>
<td>13000</td>
<td>10^6</td>
<td>10^300</td>
</tr>
<tr>
<td>10^4</td>
<td>10^4</td>
<td>10^6</td>
<td>10^8</td>
<td>10^12</td>
<td>10^3000</td>
</tr>
<tr>
<td>10^5</td>
<td>10^5</td>
<td>10^6</td>
<td>10^10</td>
<td>10^15</td>
<td>10^3000</td>
</tr>
</tbody>
</table>

how much times uses the algorithm?

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10^4</th>
<th>10^5</th>
<th>10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>log n</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>0.00002s</td>
</tr>
<tr>
<td>n</td>
<td>&lt;</td>
<td>&lt;</td>
<td>0.001s</td>
<td>0.01s</td>
<td>0.1s</td>
<td>1s</td>
</tr>
<tr>
<td>n log n</td>
<td>&lt;</td>
<td>&lt;</td>
<td>0.013s</td>
<td>0.1s</td>
<td>1s</td>
<td>10s</td>
</tr>
<tr>
<td>n^2</td>
<td>&lt;</td>
<td>0.01s</td>
<td>1s</td>
<td>100s</td>
<td>3h</td>
<td>1000h</td>
</tr>
<tr>
<td>n^3</td>
<td>0.001s</td>
<td>1s</td>
<td>1000s</td>
<td>1000h</td>
<td>100y</td>
<td>10^5y</td>
</tr>
<tr>
<td>2^n</td>
<td>0.001s</td>
<td>10^{23}y</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

1 step takes 1µs (0.000001s)
< means(< 0.001s)
> means 10^{300} year

read book 3.2

\[ \sum_{i=0}^{n} a^i = \frac{1 - a^{n+1}}{1 - a} \]

\[ \sum_{i=1}^{n} i = \frac{n(n + 1)}{2} \]

overview

- recap
- asymptotic bounds
- trees and binary trees
- heaps
- heap bubble
- building a heap
- heapsort
towards heap sort

use a tree data structure
implemented using an array

tree: definitions

(see book B5)
set of nodes with a parent-child relation
there is a unique distinguished node root
every non-root has a unique ancestor / predecessor / parent
a node may have successors / descendants / children
a node without successors is a leaf or external
a node with successors is internal

tree: more definitions

the depth of a node is the length of a path to the root
the height of a node is the maximal length of a path to a leaf
the height of a tree is the height of the root, or the maximal depth
a level or layer of a tree consists of all nodes of the same depth
the number of levels is the height of the tree plus one
binary tree: definitions

binary tree:
every node has zero, one, or two (ordered) successors

complete binary tree:
all leaves have the same depth and all depths are completely filled

almost or nearly complete binary tree:
complete binary tree where the lowest level is filled from the left up to some point

complete ⇒ almost complete ⇒ normal binary

almost complete binary trees: counting

consider an almost complete binary tree of height \( h \)

if the lowest level contains one element:
number of elements is \( n = 1 + 2 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h \)

if the lowest level is full:
number of elements is \( n = 1 + 2 + \ldots + 2^h = 2^{h+1} - 1 \)

so \( 2^h \leq n \leq 2^{h+1} - 1 \leq 2^{h+1} \)

so \( h \leq \log n \leq h + 1 \)

so \( h = \lfloor \log n \rfloor \)

this is important for the complexity of heap sort

parent-children relation in the array

\( i \) an index in the array-representation

**Algorithm** parent\((i)\):

```
return \( \lfloor i/2 \rfloor \)
```

**Algorithm** left\((i)\):

```
return \( 2i \)
```

**Algorithm** right\((i)\):

```
return \( 2i + 1 \)
```

overview

- recap
- asymptotic bounds
- trees and binary trees
- heaps
- heap bubble
- building a heap
- heapsort
heap

data structure used for sorting
and for other things priority queue

heap: definition

condition on the shape:
an almost complete binary tree

every node is labeled with a key / label from a totally ordered set

condition on the keys:
either max-heap condition on the keys:
on every path from the root to a leaf the labels / keys are non-increasing
hence max key at the root

or min-heap condition on the keys:
on every path from the root to a leaf the labels / keys are non-decreasing
hence min key at the root

max-heap: example

11 7 9
16 14 10 11 7 9
14 10
16

min-heap: example

7 9
10 16
11
7 9 11 10 16 14

for heapsort we use max-heaps
max-heap and min-heap properties

max-heap property: $H[\text{parent}(i)] \geq H[i]$

min-heap property: $H[\text{parent}(i)] \leq H[i]$

heaps: their strengths

turn an array of length $n$ into a max-heap in $O(n)$

find the largest element in a max-heap in $O(1)$

remove the largest element from a max-heap and restore in $O(\log n)$

overview

- recap
- asymptotic bounds
- trees and binary trees
- heaps
- heap bubble
- building a heap
- heapsort

bubble in heaps

we have a node with left and right heaps

we reconstruct the heap property using a down-heap bubble

we have a heap and add a labelled node

we reconstruct the heap property using a up-heap bubble
down-heap bubble or maxHeapify for max-heap

\[ \text{downMaxHeap}(H, i) \text{ (or maxHeapify) with } i \text{ a node in } H \]

left and right of \( i \) are max-heaps

- consider \( i \), its left-child \( l \) and its right-child \( r \)
- determine max of labels of \( i \), \( l \), \( r \)
- if \( i \) has the largest label then done
- if \( l \) largest label: swap labels of \( i \) and of \( l \), do downMaxHeap(H, l)
- if \( r \) largest label; swap labels of \( i \) and of \( r \), do downMaxHeap(H, r)

bubble: time complexity

time complexity of down-heap bubble

determined by height of the heap

so in \( O(\log(n)) \)

down-heap bubble: pseudo-code

Algorithm downMaxHeap(A, i):

\[ l := \text{left}(i) \]
\[ r := \text{right}(i) \]

if \( l \leq A.\text{heap-size} \) and \( A[l] > A[i] \) then

\[ \text{largest} := l \]

else

\[ \text{largest} := i \]

if \( r \leq A.\text{heap-size} \) and \( A[r] > A[\text{largest}] \) then

\[ \text{largest} := r \]

if \( \text{largest} \neq i \) then

\[ \text{swap}(A[i], A[\text{largest}]) \]

\[ \text{downMaxHeap}(A, \text{largest}) \]

overview

- recap
- asymptotic bounds
- trees and binary trees
- heaps
- heap bubble
- building a heap
- heapsort
how to build a heap?

(1) given \( n \) keys, insert them one by one
one insertion in \( O(\log n) \) (not yet discussed)
so totally in \( O(n \log n) \)

(2) using the bottom-up heap construction
then in \( O(n) \)
the latter is used in the heapsort algorithm

bottom-up heap construction: idea

we turn the binary tree with root \( i \) into a heap as follows:
assume that the binary trees with roots \( 2i \) and \( 2i + 1 \) are already heaps
with at the roots the numbers \( k \) and \( l \)
perform if necessary a swap of labels to obtain that at index \( i \) we have the
largest one
continue with down-heaping

building a heap: example

building a heap: pseudo-code

\[
\text{Algorithm } \text{buildMaxHeap}(H):
\]
\[
H.size := H.length
\]
\[
\text{for } i = \lfloor H.length/2 \rfloor \text{ downto } 1 \text{ do }
\]
\[
downMaxHeap(H, i)
\]
building a heap: complexity

\( n/2 \) calls of downMaxHeap which is in \( O(\log n) \)
so in \( O(n \log n) \)
correct but not asymptotically tight
more subtle analysis yields
worst-case time complexity of buildMaxHeap in \( O(n) \)

heapsort: idea

first part:
- turn the input-array into a max-heap
second part:
- swap the key on the root with the key on the last node
- exclude the last node from the heap
- reconstruct the heap

heapsort: pseudo-code

\( H[1 \ldots n] \) an array of integers
directly after building the heap: \( H.heap-size = H.length \)

**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(\( H, 1 \))
example heapsort

[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

heapsort: properties

why is heapsort correct?

what is the worst-case running time of heapsort?

buildMaxHeap in $O(n)$

$n - 1$ calls of downHeap

every call in $O(\log n)$

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

why is heapsort in-place?

heapsort: inventor

J.W.J. Williams in 1964

practical work 1

implement insertion sort, merge sort, and heap sort in Java

global constraints completely correct: 7 points

programming style: 1 point

annotations: 2 points

hand in via Blackboard