overview

- analysis of quicksort
- lower bound on sorting
- linear sorting algorithms
- stacks
- queues
- material
overview

- analysis of quicksort
- lower bound on sorting
- linear sorting algorithms
- stacks
- queues
- material
correctness of partition

\[ \leq \text{pivot}: \; p \ldots i \]

\[ > \text{pivot}: \; i + 1 \ldots j - 1 \]

?: \; j \ldots r - 1

for-loop terminated, then \( j = r \)
exercise

read correctness of partition and explain in your own words
running time of partition

linear in number of elements:

on input $A[p \ldots r]$ in $\Theta(n)$ with $n = r - 1 + 1$
worst-case running time of quicksort

running time of partition in $\Theta(n)$

worst-case running time if no ‘small ones’ or no ‘big ones’

worst-case running time of quicksort described by

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

$$T(n) = T(n-1) + n \in \Theta(n^2)$$

so worst-case running time of quicksort in $\Theta(n^2)$
best-case running time of quicksort

running time of partition in $\Theta(n)$

best-case running time if as many ‘small ones’ as ‘big ones’

best-case running time of quicksort described by

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \in \Theta(n \cdot \log n)$$

so best-case running time of quicksort in $\Theta(n \cdot \log n)$
suppose partition splits array in $\frac{1}{4}$ and $\frac{3}{4}$ path

$$T(n) = T\left(\frac{3}{4} n\right) + T\left(\frac{1}{4} n\right) + c \cdot n$$

recursion tree terminates at $\log_{\frac{4}{3}} n = \frac{\log 2n}{\log_2\left(\frac{4}{3}\right)}$ is roughly $\log(n)$

any constant split yields a recursion tree of roughly $\log(n)$ levels

work per level is $c \cdot n$ so running time in $\mathcal{O}(n \cdot \log n)$

any constant split yields running time in $\mathcal{O}(n \cdot \log n)$
randomized quicksort

pivot is random key from input sequence

pivot good if small-part and large-part of the partition both $< \frac{3}{4}$ of input-size

pivot bad if small-part or large-part of the partition $\geq \frac{3}{4}$ of input-size

probability of good pivot: $\frac{1}{2}$

what are good and bad pivots for 1, \ldots, 16?
randomized quicksort: expected running time

expectation for node in recursion tree on depth $i$:

\[ \frac{i}{2} \text{ ancestors are calls with good pivots} \]

hence: length array on depth $i$ is

\[ \leq \left( \frac{3}{4} \right)^{\frac{i}{2}} \cdot n \]

length 1 is reached for $i = 2 \log_\frac{4}{3} n$

hence: height is in $\mathcal{O}(\log n)$

work per depth: in $\Theta(n)$

randomized quicksort: expected running time

in case of a (possibly bad) constant split:

recursion tree in $\Theta(\log n)$, work per level in $\mathcal{O}(n)$,

best-case in $\Theta(n \cdot \log n)$,

so expected running time in $\mathcal{O}(n \cdot \log n)$

even if we have (a few) worst-case splits:

recursion tree in $\Theta(\log n)$, work per level in $\mathcal{O}(n)$,

best-case in $\Theta(n \cdot \log n)$,

so expected running time in $\mathcal{O}(n \cdot \log n)$
quicksort: inventor

Tony Hoare (1934)  Turing Award 1980
overview

- analysis of quicksort
- lower bound on sorting
- linear sorting algorithms
- stacks
- queues
- material
overview

- analysis of quicksort
- lower bound on sorting
- linear sorting algorithms
- stacks
- queues
- material
bounds

asymptotic upper bound $O$
$f$ is eventually bound from above by $g$ up to a constant

asymptotic tight bound $\Theta$
$f$ is eventually sandwiched by $g$ up to two constants

asymptotic lower bound $\Omega$
$f$ is eventually bound from below by $g$ up to a constant

(there are more bounds)
upper and lower bound

\[ f, g : \mathbb{N} \to \mathbb{R}^+ \]

asymptotic upper bound: \( f(n) \in \mathcal{O}(g(n)) \) if

\[
\exists c > 0 \in \mathbb{R} \\
\exists n_0 > 0 \in \mathbb{N} \\
\forall n \in \mathbb{N} : n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)
\]

asymptotic lower bound: \( f(n) \in \Omega(g(n)) \) if

\[
\exists c > 0 \in \mathbb{R} \\
\exists n_0 > 0 \in \mathbb{N} \\
\forall n \in \mathbb{N} : n \geq n_0 \Rightarrow f(n) \geq c \cdot g(n)
\]
lower bound on worst-case running time

we consider sorting algorithms based on comparisons $k < k'$

examples: merge sort, heapsort, quicksort

non-examples: counting sort, bucket sort

$\Omega(n \log n)$ comparisons are needed in the worst case

so merge sort, heapsort, quicksort are asymptotically optimal
lowerbound on worst-case running time: proof

we consider the decision tree
	node contains a comparison $a < b$?

leaf corresponds to a permutation of $\{1, \ldots, n\}$

every possible permutation (of total $n!$) must occur

every permutation must be reachable

path corresponds to execution of a sorting algorithm
decision tree: example

```
a < b?
b < c?
a < c?
  a < c?
    a < c?
      a < c?
        a < c?
          a < c?
            a < c?
              a < c?
                a < c?
                a < c?
                a < c?
    a < c?
      a < c?
        a < c?
          a < c?
            a < c?
              a < c?
                a < c?
                a < c?
                a < c?
  a < c?
    a < c?
      a < c?
        a < c?
          a < c?
            a < c?
              a < c?
                a < c?
                a < c?
                a < c?
  a < c?
    a < c?
      a < c?
        a < c?
          a < c?
            a < c?
              a < c?
                a < c?
                a < c?
                a < c?
  a < c?
    a < c?
      a < c?
        a < c?
          a < c?
            a < c?
              a < c?
                a < c?
                a < c?
                a < c?
  a < c?
    a < c?
      a < c?
        a < c?
          a < c?
            a < c?
              a < c?
                a < c?
                a < c?
                a < c?
```
lower bound on worst-case running time: theorem

the number of comparisons is height of the tree
we have at least $n!$ and at most $2^h$ leaves

$n! \leq 2^h$ so $h \geq \lceil \log n! \rceil$

$$2^h \geq n!$$

$$h \geq \log(n!)$$

$$= \log(1 \cdot 2 \cdot \ldots \cdot \frac{n}{2} \cdot \ldots \cdot n)$$

$$\geq \log(1 \cdot 1 \cdot \ldots \cdot 1 \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2})$$

$$= \log((\frac{n}{2})^{\frac{n}{2}})$$

$$= \frac{n}{2} \log(\frac{n}{2})$$

hence $h \in \Omega(n \log n)$

(can also be done using Stirling’s formula)
back to puzzle from Knuth

can we sort 5 elements in $\lceil \log 5! \rceil = 7$ comparisons?
overview

- analysis of quicksort
- lower bound on sorting
- linear sorting algorithms
- stacks
- queues
- material
lower bound

asymptotic lower bound on worst-case running time of comparison based sorting algorithms

$Ω(n \log n)$

we will see some (not comparison based) linear sorting algorithms with a ‘better’ lower bound
counting sort

assumption: numbers in input come from fixed range \( \{0, \ldots, k\} \).

algorithm idea: count the number of occurrences of each \( i \) from 0 to \( k \)

time complexity: in \( \Theta(n + k) \) for a input-array of length \( n \)

drawback: fixed range, and requires additional arrays of length \( m \)
counting sort: pseudo-code

input array $A$, output array $B$, range from 0 up to $k$

**Algorithm** countingSort($A$, $B$, $k$):

1. new array $C[0 \ldots k]$
2. for $i := 0$ to $k$ do
   - $C[i] := 0$
3. for $j := 1$ to $A$.length do
   - $C[A[j]] := C[A[j]] + 1$
4. for $i := 1$ to $k$ do
   - $C[i] := C[i] + C[i - 1]$
5. for $j := A$.length downto 1 do
   - $C[A[j]] := C[A[j]] - 1$
lexicographic ordering

like in a dictionary

for vectors:

$(x_1, \ldots, x_d) < (y_1, \ldots, y_d)$ if

$(x_1 < y_1)$ or

$((x_1 = y_1) \text{ and } (x_2, \ldots, x_n) < (y_2, \ldots, y_n))$
radix sort for card-sorting machine
radix sort: pseudo-code

sort per dimension using a stable sorting algorithm

Algorithm radixSort(A, d):
    for $i := 1$ to $d$ do
        use stable sort on digit $d$

1 is the lowest-order digit and $d$ is the highest-order digit
when using wrong order not correct

then goes to

13
22
11

11
22
13
radix sort: example

(7, 4, 6)  (5, 1, 5)  (2, 0, 6)  (5, 1, 4)  (2, 1, 4)
(5, 1, 4)  (2, 1, 4)  (5, 1, 5)  (7, 4, 6)  (2, 0, 6)
(2, 0, 6)  (5, 1, 4)  (2, 1, 4)  (5, 1, 5)  (7, 4, 6)
(2, 0, 6)  (2, 1, 4)  (5, 1, 4)  (5, 1, 5)  (7, 4, 6)

dimension 3
dimension 2
dimension 1
radix sort: time complexity

if the stable sorting algorithm (counting) per dimension is in $\Theta(n + k)$

then radix sort is in $\Theta(d(n + k))$
radix sort: application

sort $n$ integers in range $\{0, \ldots, k\}$ in representation with $d$ digits

use radix sort with dimension $d$

example:
range $\{0, \ldots, 8\}$ in representation with basis 3:
$00, 01, 02, 10, 11, 12, 20, 21, 22$
bucket sort

similar to counting sort

keys uniformly distributed in fixed range

put item with label $k$ in bucket array at index $k$
**bucket sort**

array $A$ with $0 \leq A[i] < 1$ for $i = 1, \ldots, n$

**Algorithm** $\text{bucketSort}(A)$:

$n := A.length$

new array $B[0 \ldots n - 1]$

for $i := 0$ to $n - 1$ do

   make $B[i]$ an empty list

for $i := 1$ to $n$ do

   insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor ]$

for $i := 0$ to $n - 1$ do

insertionSort($B[i]$)
bucket sort: example
overview

- analysis of quicksort
- lower bound on sorting
- linear sorting algorithms
- stacks
- queues
- material
stacks: properties

LIFO: last-in first-out

examples: stack of plates, memory of the undo button in an editor

any data can be on a stack

operations:
push an element onto the stack
pop the top element from the stack and return it

example:
other example: method stack in the Java Virtual Machine

Active methods, with local variables and a program counter

```java
main() {
    int i = 5;
    foo(i);
}

foo(int j) {
    int k;
    k = j+1;
    bar(k);
}

bar(int m) {
    ...
}
```

- **bar**: PC = 1, m = 6
- **foo**: PC = 3, j = 5, k = 6
- **main**: PC = 2, i = 5
Abstract Data Type (ADT)

an abstract data type (ADT) specifies
what are the operations
and what are the errors / exceptions
ADT for stacks

main operations:

- \text{push}(S, o)
  pushes element \( o \) onto the top of the stack \( S \)
- \text{pop}(S)
  removes the top element from the stack and returns it

some auxiliary operations:

- \text{size}(S)
  returns the number of elements of the stack
- \text{isEmpty}(S)
  returns a boolean indicating whether the stack is empty
- \text{top}(S)
  returns the top element of the stack (\textit{without removing it})

exceptions or errors for:

- calling \text{pop} or \text{top} to an empty stack
stacks: implementation using an array

the maximum size of the stack is determined beforehand

elements are added from left to right in the array

if the stack is empty then $S.top = 0$

below $S.top = 6$

for overflow we need an exception (which is not in the ADT)

```
S 3 2 5 4 1 4    
  1 2 3 ... 6
```
Algorithm push(S, o):
    if S.top = N then
        throw FullStackException
    else
        S.top := S.top + 1
        S[S.top] := o
Algorithm pop(S):
    if isEmpty(S) then
        throw EmptyStackException
    else
        o := S[S.top]
        S.top := S.top − 1
    return o
question: give pseudo-code for `size(S)`, `isEmpty(S)`, `top(S)`
all operations in $O(1)$ (why?)
maximum size is determined beforehand (not according to ADT)
note the errors / exceptions
overview

- analysis of quicksort
- lower bound on sorting
- linear sorting algorithms
- stacks
- queues
- material
queues: properties

FIFO: first-in first-out

dependencies: post office, printer queue

any data can be on a queue

operations:
enqueue an element at the tail of the queue
dequeue the element at the head and return it

example:
ADT for queues

main operations:
- `enqueue(Q, o)` adds element `o` at the tail of the queue
- `dequeue(Q)` removes element at the head and returns it

auxiliary operations:
- `size(Q)` returns the number of elements of the queue
- `isEmpty(Q)` returns a boolean indicating whether the queue is empty
- `front(Q)` returns the head element of the queue (without removing it)

exceptions / errors:
- call of `dequeue()` or `front()` to an empty queue
queues: implementation using circular array

the maximum size of the queue is determined beforehand

\( Q.head \) indicates position of the head (front)

\( Q.tail \) indicates position right of the tail (rear)

if the queue is empty then \( Q.head = Q.tail \)

initially, \( Q.head = Q.tail = 1 \)

in case of overflow, we need an exception
queues: implementation using circular array

normal configuration

```
Q: 1 2 3 ... h r
```

wrapped configuration

```
Q: r h
```

empty queue (head == tail)

```
Q: head
tail
```
pseudo-code for enqueue

size of the array fixed

elements are at indices $Q\.head$, $Q\.head + 1$, \ldots, $Q\.tail - 1$

check on overflow omitted

\begin{algorithm}
\begin{algorithmic}
\caption{enqueue($Q, x$)}
\State $Q[Q\.tail] := x$
\If{$Q\.tail = Q\.length$}
\State $Q\.tail := 1$
\Else
\State $Q\.tail := Q\.tail + 1$
\EndIf
\end{algorithmic}
\end{algorithm}
Algorithm dequeue(Q):
   \( x := Q[Q.head] \)
   \textbf{if} \ Q.head = Q.length \textbf{then}
      \( Q.head := 1 \)
   \textbf{else}
      \( Q.head := Q.head + 1 \)
queues: varia

give pseudo-code for size(Q), isEmpty(Q), front(Q)

operations are all in \( \mathcal{O}(1) \)

maximum size determined beforehand (not according to ADT)

note the exceptions / errors
overview

- analysis of quicksort
- lower bound on sorting
- linear sorting algorithms
- stacks
- queues
- material
extra materiaal

- wiki over deques