Searching and Sorting Algorithms, Complexity Analysis

Searching Algorithms

- General definition
  - Locate an element x in a list of distinct elements $a_1, a_2, \ldots, a_n$, or determine that it is not in the list
- Linear search

Algorithm 2: Linear search
Input: unsorted sequence $a_1, a_2, \ldots, a_n$
  position of target value x
Output: subscript of entry equal to target value; 0 if not found
Initialize: $i \leftarrow 1$
while ($i \leq n$ **and** $x \neq a_i$)
  $i \leftarrow i + 1$
if $i \leq n$ **then** *location* $\leftarrow i$ **else** *location* $\leftarrow 0$

Searching Algorithms

– Binary search
- Requires input sequence to be sorted

General Idea:

Find 9 in the sequence: <1, 4, 6, 9, 10, 14>

left mid right
<1, 4, 6, 9, 10, 14>

$9 > a_{mid}$?
Yes (9 > 6)

left mid right
<1, 4, 6, 9, 10, 14>

$9 > a_{mid}$?
No (9 < 10)

left mid right
<1, 4, 6, 9, 10, 14> …

Searching Algorithms

– Binary search

Algorithm 3: Binary search
Input: sorted sequence $a_1, a_2, \ldots, a_n$
  position of target value x
Output: subscript of entry equal to target value; 0 if not found
Initialize: *left* $\leftarrow 1$; *right* $\leftarrow n$
while (*left* < *right*)
  mid $\leftarrow \lfloor (left + right) / 2 \rfloor$
  if $x > a_{mid}$ **then** *left* $\leftarrow$ *mid* + 1 **else** *right* $\leftarrow$ *mid*
if $x = a_{left}$ **then** *location* $\leftarrow$ *left* **else** *location* $\leftarrow 0$

Complexity Analysis

- Usually time complexity considered
- Space complexity can also be considered
- RAM Model
  - Constant time basic operations (add, sub, load, store…)
- Worst-case complexity measure
  - Estimates the time required for the most time-consuming input of each size
- Average-case complexity measure
  - Estimates the average time required for input of each size
  - Not always easy to see what is the average-case

Complexity Analysis

Algorithm 2: Linear search
Initialize: i ← 1
while ($i \leq n$ **and** $x \neq a_i$)
    $i \leftarrow i + 1$
if $i \leq n$ **then** *location* ← *i* **else** *location* ← *0*

Worst-case: element is at the end of the sequence or is not present n iterations of loop · $c_1 + c_2$ => $\Theta(n)$

Avrg-case: element is at the middle of the sequence

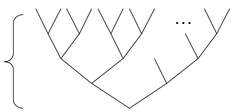
n/2 iterations of loop · $c_1 + c_2$ => $\Theta(n)$

Complexity Analysis

Algorithm 3: Binary search
Initialize: left ← 1; *right* ← *n*
while (*left* < *right*)
    mid ← $\lfloor(left + right) / 2\rfloor$
    if $x > a_{mid}$ **then** *left* ← *mid + 1* **else** *right* ← *mid*
if $x = a_{left}$ **then** *location* ← *left* **else** *location* ← *0*

n elements

height = k



Average & worst-case analysis:

k iteration of while loop · $c_1 + c_2$ => $\Theta(k) = \Theta(\log_2 n)$

Sorting Algorithms

- General definition
  - Putting a number of elements into a list in which the elements are in increasing order
- Input:
  - A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
- Output:
  - A permutation (reordering) $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Sorting Algorithms

- Insertion sort
- General idea:
  - Same idea as what you do when cards are distributed
  - Your left hand is initially empty
  - Until all cards have been distributed
    - Take a card with right hand and insert it at the right position in left hand

Sorting Algorithms

- Insertion sort

Algorithm 3: Insertion sort
Input: unsorted sequence $a_1, a_2, \ldots, a_n$
Output: sorted sequence $a_1, a_2, \ldots, a_n$

```
for j ← 2 to n                          distribute all cards
        key ← a_j
        i ← j - 1
        while i > 0 and a_i > a_j        insert at right position in left hand
                a_{i+1} ← a_i
                i ← i - 1
        a_i ← key
```


Insertion Sort - example

- j ↓ <3, 18, 4, 10, 7> <3, 18, 4, 10, 7>
- j ↓ <3, 18, 4, 10, 7> <3, 4, 18, 10, 7>
- j ↓ <3, 4, 18, 10, 7> <3, 4, 10, 18, 7>
- j ↓ <3, 4, 10, 18, 7> <3, 4, 7, 10, 18>

Correctness proof of insertion sort

- Must prove that:
  - The algorithm terminates
  - The algorithm sorts the input sequence
- Termination:
  - All operations of the algorithm take a finite amount of time
  - The algorithm executes a bounded number of loop iterations

Correctness proof of insertion sort

– The algorithm sorts the input sequence

Algorithm 3: Insertion sort

for $j \leftarrow 2$ **to** n
    $key \leftarrow a_j$
    $i \leftarrow j - 1$
    while $i > 0$ **and** $a_i > a_j$
        $a_{i+1} \leftarrow a_i$
        $i \leftarrow i - 1$
    $a_i \leftarrow key$

- Prove a loop invariant:
  - $a_1 \ldots a_j$ sorted at end of iteration j

Correctness proof of insertion sort

Algorithm 3: Insertion sort

for $j \leftarrow 2$ **to** n
    $key \leftarrow a_j$
    $i \leftarrow j - 1$
    while $i > 0$ **and** $a_i > a_j$
        $a_{i+1} \leftarrow a_i$
        $i \leftarrow i - 1$
    $a_i \leftarrow key$

– Prove loop invariant by induction
- Base step: j = 2
- Induction step: if true for $2 \le k < j$ then true for j

Correctness proof of insertion sort

Algorithm 3: Insertion sort

for $j \leftarrow 2$ **to** n
    $key \leftarrow a_j$
    $i \leftarrow j - 1$
    while $i > 0$ **and** $a_i > a_j$
        $a_{i+1} \leftarrow a_i$
        $i \leftarrow i - 1$
    $a_i \leftarrow key$

– Base step (j = 2): $a_1 \ldots a_2$
- Property holds by condition of while loop and last line

Correctness proof of insertion sort

Algorithm 3: Insertion sort

for $j \leftarrow 2$ **to** n
    $key \leftarrow a_j$
    $i \leftarrow j - 1$
    while $i > 0$ **and** $a_i > a_j$
        $a_{i+1} \leftarrow a_i$
        $i \leftarrow i - 1$
    $a_i \leftarrow key$

– Induction step:
- Property holds by condition of while loop and last line

Correctness proof of insertion sort

- We proved that $a_1 \ldots a_j$ is sorted at end of iteration j
- The last iteration is when j = n
- Consequently, when the algorithm terminates, $a_1 \ldots a_j$ is sorted

Sorting Algorithms

- Bubble Sort

Algorithm 4: Bubble sort
Input: unsorted sequence $a_1, a_2, \ldots, a_n$
Output: sorted sequence $a_1, a_2, \ldots, a_n$
for $i \leftarrow 1$ **to** n
  for $j \leftarrow n$ **to** $i + 1$
    if $a_j < a_{j-1}$ **then** interchange a_j and a_{j-1}

Bubble sort - example

- <18, 4, 10, 7> <4, 18, 7, 10>
- < 4, 18, 7, 10 > < 4, 7, 18, 10>
- < 4, 7, 18, 10 > < 4, 7, 10, 18>
- < 4, 7, 10, 18 > <4, 7, 10, 18>

Complexity Analysis

Algorithm 3: Insertion sort

for $j \leftarrow 2$ **to** n
  $key \leftarrow a_j$
  $i \leftarrow j - 1$
  while $i > 0$ **and** $a_i > a_j$
    $a_{i+1} \leftarrow a_i$
    $i \leftarrow i - 1$
  $a_i \leftarrow key$

Worst-case: input is sorted in reverse order

$$\sum_{j=2}^{n}\sum_{i=1}^{j-1} c = c + 2c + \ldots + (n-1)c = \frac{(n-1)^2}{2}c \Rightarrow \Theta(n^2)$$

Complexity Analysis

Algorithm 3: Insertion sort

```
for j ← 2 to n
      key ← a_j
      i ← j - 1
      while i > 0 and a_i > a_j
            a_{i+1} ← a_i
            i ← i - 1
      a_i ← key
```

Average-case: half of the elements in $a_1..a_{j-1}$ are greater than a_j

$$\sum_{j=2}^{n}\sum_{i=1}^{\lceil (j-1)/2\rceil} c = c + c + 2c + 2c + \ldots + \left\lceil \frac{n-1}{2} \right\rceil c \approx \frac{n+1}{2} \cdot \frac{n-1}{2} \Rightarrow \Theta(n^2)$$

Complexity Analysis

Algorithm 4: Bubble sort

for $i \leftarrow 1$ **to** n
    for $j \leftarrow n$ **to** $i+1$
        if $a_j < a_{j-1}$ **then** interchange a_j and a_{j-1}

Average & worst-case:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} c = (n-1)c + (n-2)c + \ldots + c = \frac{(n-1)^2}{2}c \quad \Rightarrow \Theta(n^2)$$

Summary

Algorithm	Avrg-case	Worst-case
Linear search	$\Theta(n)$	$\Theta(n)$
Binary search	$\Theta(log_2(n))$	$\Theta(log_2(n))$
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$

Complexity of algorithms

– Common terminology

Complexity	***Terminology***
$O(1)$	Constant complexity
$O(\log n)$	Logarithmic complexity
$O(n)$	Linear complexity
$O(n \log n)$	$n \log n$ complexity
$O(n^b)$	Polynomial complexity
$O(b^n)$, $b > 1$	Exponential complexity
$O(n!)$	Factorial complexity