Fundamental Algorithms

Chapter 2: Sorting

Michael Bader

Winter 2013/14

Part I

Simple Sorts

The Sorting Problem

Definition

Sorting is required to order a given sequence of elements, or more precisely:

Input : a sequence of n elements $a_1, a_2, \ldots, a_n$

Output : a permutation (reordering) $a'_1, a'_2, \ldots, a'_n$ of the input sequence, such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- we will assume the elements $a_1, a_2, \ldots, a_n$ to be integers (or any element/data type on which a total order $\leq$ is defined)
- a sorting algorithm may output the permuted data or also the permuted set of indices

Insertion Sort

Idea: sorting by inserting

- successively generate ordered sequences of the first j numbers: $j = 1, j = 2, \ldots, j = n$
- in each step, $j \to j + 1$, one additional integer has to be inserted into an already ordered sequence

Data Structures:

- an array A[1..n] that contains the sequence a_1 (in A[1]), . . . , a_n (in A[n]).
- numbers are sorted **in place**:
  output sequence will be stored in A itself
  (hence, content of A is changed)

Insertion Sort – Implementation

```
InsertionSort(A:Array[1..n]) {

  for j from 2 to n {
  // insert A[j] into sequence A[1..j−1]

     key := A[j];

     i := j−1; // initialize i for while loop
     while i>=1 and A[i]>key {
        A[i+1] := A[i];
        i := i−1;
     }
     A[i+1] := key;
  }
}
```

Correctness of InsertionSort

Loop invariant:

Before each iteration of the for-loop, the subarray A[1..j-1] consists of all elements originally in A[1..j-1], but in sorted order.

Initialization:

- loops starts with j=2;
  hence, A[1..j-1] consists of the element A[1] only
- A[1] contains only one element, A[1], and is therefore sorted.

Correctness of InsertionSort

Loop invariant:

Before each iteration of the for-loop, the subarray A[1..j-1] consists of all elements originally in A[1..j-1], but in sorted order.

Maintenance:

- assume that the while loop works correctly (or prove this using an additional loop invariant):
  - after the while loop, i contains the largest index for which A[i] is smaller than the key
  - A[i+2..j] contains the (sorted) elements previously stored in A[i+1..j-1]; also: A[i+1] and all elements in A[i+2..j] are $\geq$ key
- the key value, A[j], is thus correctly inserted as element A[i+1] (overwrites the duplicate value A[i+1])
- after execution of the loop body, A[1..j] is sorted
- thus, before the next iteration (j:=j+1), A[1..j-1] is sorted

Correctness of InsertionSort

Loop invariant:

Before each iteration of the for-loop, the subarray A[1..j-1] consists of all elements originally in A[1..j-1], but in sorted order.

Termination:

- The for-loop terminates when j exceeds n (i.e., j=n+1)
- Thus, at termination, A[1 .. (n+1)-1] = A[1..n] is sorted and contains all original elements

Insertion Sort – Number of Comparisons

```
InsertionSort(A:Array[1..n]) {
   for j from 2 to n {
      key := A[j];
      i := j-1;
      while i>=1 and A[i]>key {
         A[i+1] := A[i];
         i := i-1;
      }
      A[i+1] := key;
   }
}
```

n-1 iterations

t_j iterations
$\rightarrow$ t_j comparisons
A[i] $>$ key

$$\Rightarrow \sum_{j=2}^{n} t_j \text{ comparisons}$$

Insertion Sort – Number of Comparisons (2)

- counted number of comparisons: $T_{\mathrm{IS}} = \sum_{j=2}^{n} t_j$
- where t_j is the number of iterations of the while loop (which is, of course, unknown)
- good estimate for the run time, if the comparison is the most expensive operation (note: replace "i $>$=1" by for loop)

Analysis

- what is "best case"?
- what "worst case"?

Insertion Sort – Number of Comparisons (2)

- counted number of comparisons: $T_{\text{IS}} = \sum_{j=2}^{n} t_j$
- where t_j is the number of iterations of the while loop (which is, of course, unknown)
- good estimate for the run time, if the comparison is the most expensive operation (note: replace "i>=1" by for loop)

Analysis of the "best case":

- in the best case, $t_j = 1$ for all j
- happens only, if A[1..n] is already sorted

$$\Rightarrow T_{\text{IS}}(n) = \sum_{j=2}^{n} 1 = n - 1 \in \Theta(n)$$

Insertion Sort – Number of Comparisons (2)

- counted number of comparisons: $T_{\text{IS}} = \sum_{j=2}^{n} t_j$
- where t_j is the number of iterations of the while loop (which is, of course, unknown)
- good estimate for the run time, if the comparison is the most expensive operation (note: replace "i$>$=1" by for loop)

Analysis of the "worst case":

- in the worst case, $t_j = j - 1$ for all j
- happens, if A[1..n] is already sorted in opposite order

$$\Rightarrow T_{\text{IS}}(n) = \sum_{j=2}^{n} (j-1) = \frac{1}{2} n(n-1) \in \Theta(n^2)$$

Insertion Sort – Number of Comparisons (2)

- counted number of comparisons: $T_{\mathrm{IS}} = \sum_{j=2}^{n} t_j$
- where t_j is the number of iterations of the while loop (which is, of course, unknown)
- good estimate for the run time, if the comparison is the most expensive operation (note: replace “i$>$=1” by for loop)

Analysis of the “average case”:

- best case analysis: $T_{\mathrm{IS}}(n) \in \Theta(n)$
- worst case analysis: $T_{\mathrm{IS}}(n) \in \Theta(n^2)$

$\Rightarrow$ What will be the ”typical” (average, expected) case?

Running Time and Complexity

"Run(ning)Time"

- the notation $T(n)$ suggest a "time", such as run(ning) time of an algorithm, which depends on the input (size) n
- in practice: we need a precise model how long each operation of our programmes takes $\rightarrow$ very difficult on real hardware!
- we will therefore determine the number of operations that determine the run time, such as:
  - number of comparisons (sorting, e.g.)
  - number of arithmetic operations (Fibonacci, e.g.)
  - number of memory accesses

"Complexity"

- characterises how the run time depends on the input (size), typically expressed in terms of the Θ-notation
- "algorithm xyz has linear complexity" $\rightarrow$ run time is $\Theta(n)$

Average Case Complexity

Definition (expected running time)

Let $X(n)$ be the set of all possible input sequences of length n, and let $P\colon X(n) \to [0,1]$ be a probability function such that $P(x)$ is the probability that the input sequence is x.
Then, we define

$$\bar{T}(n) = \sum_{x \in X(n)} P(x)T(x)$$

as the **expected running time** of the algorithm.

Comments:

- we require an exact probability distribution (for InsertionSort, we could assume that all possible sequences have the same probability)
- we need to be able to determine $T(x)$ for any sequence x (much too laborious to determine)

Average Case Complexity (2)

Heuristic estimate:

- we assume that we need $\frac{j}{2}$ steps in every iteration:

$$\Rightarrow \bar{T}_{\mathrm{IS}}(n) \overset{(?)}{\approx} \sum_{j=2}^{n} \frac{j}{2} = \frac{1}{2} \sum_{j=2}^{n} j \in \Theta(n^2)$$

- note: $\frac{j}{2}$ isn't even an integer...
- **Just considering the number of comparisons of the "average case" can lead to quite wrong results!**

in general $\quad E(T(n)) \neq T(\text{“}E(n)\text{”})$

Bubble Sort

```
BubbleSort(A:Array[1..n]) {
    for i from 1 to n do {
        for j from n downto i+1 do {
            if A[j] < A[j−1]
            then exchange A[j] and A[j−1]
        }
    }
}
```

Basic ideas:

- compare neighboring elements only
- exchange values if they are not in sorted order
- repeat until array is sorted (here: pessimistic loop choice)

Bubble Sort – Homework

Prove correctness of Bubble Sort:

- find invariant for i-loop
- find invariant for j-loop

Number of comparisons in Bubble Sort:

- best/worst/average case?

Part II

Mergesort and Quicksort

Mergesort

Basic Idea: divide and conquer

- **Divide** the problem into two (or more) subproblems:
  $\rightarrow$ split the array into two arrays of equal size
- **Conquer** the subproblems by solving them recursively:
  $\rightarrow$ sort both arrays using the sorting algorithm
- **Combine** the solutions of the subproblems:
  $\rightarrow$ merge the two sorted arrays to produce the entire sorted array

Combining Two Sorted Arrays: Merge

```
Merge (L:Array[1..p], R:Array[1..q], A:Array[1..n]) {
// merge the sorted arrays L and R into A (sorted)
// we presume that n=p+q
   i:=1; j:=1:
   for k from 1 to n do {
      if i > p
         then { A[k]:=R[j]; j=j+1; }
      else if j > q
         then { A[k]:=L[i]; i:=i+1; }
      else if L[i] < R[j]
         then { A[k]:=L[i]; i:=i+1; }
         else { A[k]:=R[j]; j:=j+1; }
   }
}
```

Correctness and Run Time of Merge

Loop invariant:

Before each cycle of the for loop:

- A contains the k-1 smallest elements of L and R combined;
- L[i] and R[j] are the smallest elements of L and R that have not been copied to A yet
  (L[1..i-1] and R[1..j-1] have been merged to A)

Run time:

$$T_{\text{Merge}}(n) \in \Theta(n)$$

- for loop will be executed exactly *n* times
- each loop contains constant number of commands:
  - exactly 1 copy statement
  - exactly 1 increment statement
  - 1–3 comparisons

MergeSort

```
MergeSort(A:Array[1..n]) {
    if  n > 1 then {
        m := floor (n/2);
        create array L [1... m];
        for  i  from 1 to m do { L[i]  :=  A[i ];  }

        create array R [1... n−m];
        for  i  from 1 to n−m do { R[i] := A[m+i]; }

        MergeSort(L);
        MergeSort(R);

        Merge(L,R,A);
    }
}
```

Number of Comparisons in MergeSort

- Merge performs $c \cdot n$ comparisons on n elements
- MergeSort itself does not contain any comparisons between elements; all comparisons done in Merge

$\Rightarrow$ number of comparisons for the entire MergeSort algorithms can be specified by a recurrence:

$$T_{\mathrm{MS}}(n) = \begin{cases} 0 & \text{if} \quad n \leq 1 \\ T_{\mathrm{MS}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{\mathrm{MS}}\left(n - \left\lfloor \frac{n}{2} \right\rfloor\right) + cn & \text{if} \quad n \geq 2 \end{cases}$$

Number of Comparisons in MergeSort (2)

Assume $n = 2^k$, c constant:

$$\begin{aligned}
T_{\mathrm{MS}}(2^k) &\leq T_{\mathrm{MS}}\left(2^{k-1}\right) + T_{\mathrm{MS}}\left(2^{k-1}\right) + c \cdot 2^k \\
&\leq 2\,T_{\mathrm{MS}}\left(2^{k-1}\right) + 2^k c \\
&\leq 2^2\,T_{\mathrm{MS}}\left(2^{k-2}\right) + 2 \cdot 2^{k-1} c + 2^k c \\
&\leq \ldots \\
&\leq 2^k\,T_{\mathrm{MS}}\left(2^0\right) + 2^{k-1} \cdot 2^1 c + \ldots + 2^j \cdot 2^{k-j} c \\
&\qquad\qquad + \ldots + 2 \cdot 2^{k-1} c + 2^k c \\
&\leq \sum_{j=1}^{k} 2^k c = ck \cdot 2^k = cn \log_2 n \in O(n \log n)
\end{aligned}$$

Quicksort

Basic Idea: divide and conquer

- **Divide** the input array A[p..r] into parts A[p..q] and A[q+1 .. r], such that every element in A[q+1 .. r] is larger than all elements in A[p .. q].
- **Conquer:** sort the two arrays A[p..q] and A[q+1 .. r]
- **Combine:** if the divide and conquer steps are performed in place, then no further combination step is required.

Partitioning using a **pivot element**:

- all elements that are smaller than the pivot element should go into the “smaller” partition (A[p..q])
- all elements that are larger than the pivot element should go into the “larger” partition (A[q+1..r])

Partitioning the Array (Hoare's Algorithm)

```
Partition (A:Array[p..r]) : Integer {
  // x is the pivot (chosen as first element):
  x := A[p];
  // partitions grow towards each other
  i := p; j := r; // (partition boundaries)
  while true do { // i<j: partitions haven't met yet
    // leave large elements in right partition
    while A[j]>x do { j:=j−1; };
    // leave small elements in left partition
    while A[i]<x do { i:=i+1; };
    // swap the two first "wrong" elements
    if i < j then {
      exchange A[i] and A[j];
      i:=i+1; j:=j−1;
    } else return j;
  }
}
```

Time Complexity of Partition

How many statements are executed by the nested while loops?

- monitor increments/decrements of i and j
- after $n := r - p$ increments/decrements, i and j have the same value

$\Rightarrow$ $\Theta(n)$ comparisons with the pivot

$\Rightarrow$ $O(n)$ element exchanges

Hence: $T_{\text{Part}}(n) \in \Theta(n)$

Implementation of QuickSort

```
QuickSort (A:Array[p..r ])
{
   if  p>=r then return;
   // only proceed, if A has at least 2 elements:
   q :=  Partition (A);
   QuickSort (A[p..q ]);
   QuickSort (A[q+1..r ]);
}
```

Homework:

- prove correctness of Partition
- prove correctness of QuickSort

Time Complexity of QuickSort

Best Case:

- assume that all partitions are split exactly into two halves:

$$T_{\mathrm{QS}}^{\mathrm{best}}(n) = 2T_{\mathrm{QS}}^{\mathrm{best}}\left(\frac{n}{2}\right) + \Theta(n)$$

- analogous to MergeSort:

$$T_{\mathrm{QS}}^{\mathrm{best}}(n) \in \Theta(n \log n)$$

Worst Case:

- Partition will always produce one partition with only 1 element:

$$\begin{aligned} T_{\mathrm{QS}}^{\mathrm{worst}}(n) &= T_{\mathrm{QS}}^{\mathrm{worst}}(n-1) + T_{\mathrm{QS}}^{\mathrm{worst}}(1) + \Theta(n) \\ &= T_{\mathrm{QS}}^{\mathrm{worst}}(n-1) + \Theta(n) = T_{\mathrm{QS}}^{\mathrm{worst}}(n-2) + \Theta(n-1) + \Theta(n) \\ &= \ldots = \Theta(1) + \ldots + \Theta(n-1) + \Theta(n) \in \Theta(n^2) \end{aligned}$$

Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?
  $\rightarrow$ partition sizes always 1 and n-1 $\Rightarrow \Theta(n^2)$
- A is sorted in reverse order?
  $\rightarrow$ partition sizes always 1 and n-1 $\Rightarrow \Theta(n^2)$
- one partition has always at most a elements (for a fixed a)?
  $\rightarrow$ same complexity as $a = 1 \Rightarrow \Theta(n^2)$
- partition sizes are always $n(1-a)$ and na with $0 < a < 1$?
  $\rightarrow$ same complexity as best case $\Rightarrow \Theta(n \log n)$

Questions:

- What happens in the “usual” case?
- Can we force the best case?

Randomized QuickSort

```
RandPartition ( A: Array [p..r] ): Integer {
    // choose random integer i between p and r
    i := rand(p,r);
    // make A[i] the (new) Pivot element:
    exchange A[i] and A[p];
    // call Partition with new pivot element
    q := Partition (A);
    return q;
}

RandQuickSort ( A:Array [p..r] ) {
    if p >= r then return;
    q := RandPartition(A);
    RandQuickSort (A[p...q]);
    RandQuickSort (A[q+1 ..r]);
}
```

Time Complexity of RandQuickSort

Best/Worst-case complexity?

- RandQuickSort may still produce the worst (or best) partition in each step
- worst case: $\Theta(n^2)$
- best case: $\Theta(n \log n)$

However:

- it is not determined which input sequence (sorted order, reverse order) will lead to worst case behavior (or best case behavior);
- any input sequence might lead to the worst case or the best case, depending on the random choice of pivot elements.

Thus: only the **average-case complexity** is of interest!

Average Case Complexity of RandQuickSort

Assumptions:

- we compute $\bar{T}_{\text{RQS}}(A)$,
  i.e., the expected run time of RandQuickSort for a given input A
- `rand(p,r)` will return uniformly distributed random numbers
  (all pivot elements have the same probability)
- all elements of A have different size: A[i] $\neq$ A[j]

Basic Idea:

- let k be the number of elements that are smaller than the randomly chosen pivot element ($\Rightarrow 0 \leq k < n$)
- for $0 < k < n$, the partition sizes will be k and $n - k$
- for $k = 0$, the partition sizes will be 1 and $n - 1$
- the probability of $k = 0, \ldots, n - 1$ is always $\frac{1}{n}$.

Average Case Complexity of RandQuickSort (2)

The expected runtime of RandQuickSort is

$$\begin{aligned}\bar{T}_{\text{RQS}}(n) &= \sum_{j=0}^{n-1} P(k=j)\, T_{\text{RQS}}(n|_{k=j}) \\ &= \frac{1}{n}\left(\left(T_{\text{RQS}}(1) + T_{\text{RQS}}(n-1)\right) + \sum_{j=1}^{n-1} \left(T_{\text{RQS}}(j) + T_{\text{RQS}}(n-j)\right) \right) + \Theta(n)\end{aligned}$$

- Note: all $T_{\text{RQS}}(j)$ depend on the choice of the next pivot elements
- simplifying assumption: $T_{\text{RQS}}(j) = \bar{T}_{\text{RQS}}(j)$

Average Case Complexity of RandQuickSort (3)

As $\bar{T}_{\text{RQS}}(1) \in \Theta(1)$ and $\bar{T}_{\text{RQS}}(n-1) \in \Theta(n^2)$ (worst case analysis):

$$\frac{1}{n}\left(T_{\text{RQS}}(1) + T_{\text{RQS}}(n-1)\right) \in O(n)$$

Thus

$$\begin{aligned}
\bar{T}_{\text{RQS}}(n) &= \frac{1}{n}\left(\sum_{j=1}^{n-1}\left(\bar{T}_{\text{RQS}}(j) + \bar{T}_{\text{RQS}}(n-j)\right)\right) + \Theta(n) \\
&= \frac{2}{n}\sum_{j=1}^{n-1}\bar{T}_{\text{RQS}}(j) + \Theta(n)
\end{aligned}$$

Solve recurrence equation to obtain:

$$\bar{T}_{\text{RQS}} \in \Theta(n \log n)$$

Part III

Outlook: Optimality of Comparison Sorts

Are Mergesort and Quicksort optimal?

Definition

Comparison sorts are sorting algorithms that use only comparisons (i.e. tests as $\leq, =, >, \dots$) to determine the relative order of the elements.

Examples:

- InsertSort, BubbleSort
- MergeSort, (Randomised) Quicksort

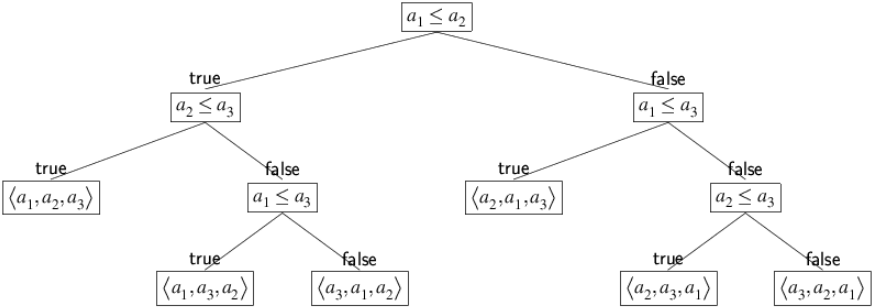
Question:

Is $T(n) \in \Theta(n \log n)$ the best we can get (in the worst/average case)?

Decision Trees

Definition

A **decision tree** is a binary tree in which each internal node is annotated by a comparison of two elements.
The leaves of the decision tree are annotated by the respective permutations that will put an input sequence into sorted order.



Decision Trees – Properties

Each comparison sort can be represented by a decision tree:

- a path through the tree represents a sequence of comparisons
- sequence of comparisons depends on results of comparisons
- can be pretty complicated for Mergesort, Quicksort, ...

A decision tree can be used as a comparison sort:

- if every possible permutation is annotated to at least one leaf of the tree!
- if (as a result) the decision tree has at least n! (distinct) leaves.

A Lower Complexity Bound for Comparison Sorts

- A binary tree of height h (h the length of the longest path) has at most 2^h leaves.
- To sort n elements, the decision tree needs $n!$ leaves.

Theorem

Any decision tree that sorts n elements has height $\Omega(n \log n)$.

Proof:

- h comparisons in the worst case are equivalent to a decision tree of height h
- with h comparisons, we can sort n elements (at best), if

$$n! \leq 2^h \quad \Leftrightarrow \quad h \geq \log(n!) \in \Omega(n \log n)$$

- because:

$$h \geq log(n!) \geq \log\left(n^{n/2}\right) = \frac{n}{2} \log n$$

Optimality of Mergesort and Quicksort

Corollaries:

- MergeSort is an optimal comparison sort in the worst/average case
- QuickSort is an optimal comparison sort in the average case

Consequences and Alternatives:

- comparison sorts can be faster than MergeSort, but only by a constant factor
- comparison sorts can not be asymptotically faster
- sorting algorithms might be faster, if they can exploit additional information on the size of elements
- examples: **BucketSort**, CountingSort, RadixSort

Bucket Sort

Basic Ideas and Assumptions:

- pre-sort numbers in buckets that contain all numbers within a certain interval
- hope (assume) that input elements are evenly distributed and thus uniformly distributed to buckets
- sort buckets and concatenate them

Requires “Buckets”:

- can hold arbitrary numbers of elements
- can insert elements efficiently: in $O(1)$ time
- can concatenate buckets efficiently: in $O(1)$ time
- remark: linked lists will do

Implementation of BucketSort

```
BucketSort (A:Array[1..n]) {

    Create Array B[0..n−1] of Buckets;
    // assume all Buckets B[i] are empty at first

    for i from 1 to n do {
        insert A[i] into Bucket B[floor(n * A[i])];
    }

    for i from 0 to n−1 do {
        sort Bucket B[i] ;
    }

    concatenate Buckets B[0], B[1], ..., B[n−1] into A
}
```

Number of Operations of BucketSort

Operations:

- n operations to distribute n elements to buckets
- plus effort to sort all buckets

Best Case:

- if each bucket gets 1 element, then $\Theta(n)$ operations are required

Worst Case:

- if one bucket gets all elements, then $T(n)$ is determined by the sorting algorithm for the buckets

Bucketsort – Average Case Analysis

- probability that bucket i contains k elements:

$$P(n_i = k) = \binom{n}{k} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k}$$

- expected mean and variance for such a distribution:

$$E[n_i] = n \cdot \frac{1}{n} = 1 \qquad \mathrm{Var}[n_i] = n \cdot \frac{1}{n}\left(1 - \frac{1}{n}\right) = \left(1 - \frac{1}{n}\right)$$

- InsertionSort for buckets $\Rightarrow \leq cn^2 \in O(n_i^2)$ operations per bucket
- expected operations to sort one bucket:

$$\bar{T}(n_i) \leq \sum_{k=0}^{n-1} P(n_i = k) \cdot ck^2 = cE[n_i^2]$$

Bucketsort – Average Case Analysis (2)

- theorem from statistics:

$$E[X^2] = E[X]^2 + \mathrm{Var}(X)$$

- expected operations to sort one bucket:

$$\bar{T}(n_i) \leq cE[n_i^2] = c\left(E[n_i]^2 + \mathrm{Var}[n_i]\right) = c\left(1^2 + 1 - \frac{1}{n}\right) \in \Theta(1)$$

- expected operations to sort all buckets:

$$\bar{T}(n) = \sum_{i=0}^{n-1} \bar{T}(n_i) \leq c \sum_{i=0}^{n-1} \left(2 - \frac{1}{n}\right) \in \Theta(n)$$

(note: expected value of the sum is the sum of expected values)