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 \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \).

- elements \( a_1, a_2, \ldots, a_n \) can be numbers; alternatively any type of element on which a total order \( \leq \) is defined
- a sorting algorithm is also allowed to output 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, $j \rightarrow j + 1$, one additional integer has to be inserted into an already ordered sequence

Data Structures:

- numbers are sorted in place: output sequence will be stored in $A$ itself (hence, content of $A$ is changed)
Insertion Sort – Implementation

\[\text{InsertionSort}(A: \text{Array}[1..n]) \{\]

\[\text{for } j \text{ from } 2 \text{ to } n \{\]
\[\quad /\!/ \text{ insert } A[j] \text{ into sequence } A[1..j-1]\]

\[\text{key} := A[j];\]

\[i := j-1; \quad /\!/ \text{initialize } i \text{ for while loop}\]
\[\text{while } i \geq 1 \text{ and } A[i] > \text{key} \{\]
\[\quad A[i+1] := A[i];\]
\[\quad i := i-1;\]
\[\}
\[A[i+1] := \text{key};\]
\}
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
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
  - $A[i+1..j]$ contains the elements previously stored in $A[i..j-1]$ maintaining the order (all elements in $A[i+1..j]$ are $\geq$ key)
- the key value, $A[j]$, is thus correctly inserted as element $A[i+1]$ (overwrites the duplicate value $A[i]$)
- 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

\[ \sum_{j=2}^{n} t_j \text{ comparisons} \]
Insertion Sort – Number of Comparisons (2)

- counted number of comparisons: \( T_{IS} = \sum_{j=2}^{n} t_j \)
- 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”?  

M. Bader: Fundamental Algorithms
Chapter 2: Sorting, Winter 2011/12
Insertion Sort – Number of Comparisons (2)

- counted number of comparisons: $T_{IS} = \sum_{j=2}^{n} t_j$
- 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 $\geq 1$” 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

$$\Rightarrow T_{IS}(n) = \sum_{j=2}^{n} 1 = n - 1 \in \Theta(n)$$
Insertion Sort – Number of Comparisons (2)

- counted number of comparisons: \( T_{IS} = \sum_{j=2}^{n} t_j \)
- 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 “\( i \geq 1 \)” 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

\[
\Rightarrow T_{IS}(n) = \sum_{j=2}^{n} (j - 1) = \frac{1}{2} n(n - 1) \in \Theta(n^2)
\]
Insertion Sort – Number of Comparisons (2)

- counted number of comparisons: \( T_{IS} = \sum_{j=2}^{n} t_j \)
- 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 “\(i \geq 1\)” by for loop)

Analysis of the “average case”:
- best case analysis: \( T_{IS}(n) \in \Theta(n) \)
- worst case analysis: \( T_{IS}(n) \in \Theta(n^2) \)

\( \Rightarrow \) What will be the ”typical” (average, expected) case?
Outlook: Average Case Complexity

Definition (expected run time)

Let $X(n)$ be the set of all possible input sequences of length $n$, and let $P: X(n) \rightarrow [0, 1]$ 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)
Heuristic estimate:

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

\[
\Rightarrow T_{IS}(n) \approx \sum_{j=2}^{n} \frac{j}{2} = \frac{1}{2} \sum_{j=2}^{n} j \in \Theta(n^2)
\]

Note: \( j^2 \) isn't even an integer...

Just considering the number of comparisons of the "average case" can lead to quite wrong results!
Outlook: Average Case Complexity (2)

Heuristic estimate:

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

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

- note: $\frac{i}{2}$ isn’t even an integer...
Outlook: Average Case Complexity (2)

Heuristic estimate:

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

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

- note: $\frac{i}{2}$ isn’t even an integer...

- Just considering the number of comparisons of the “average case” can lead to quite wrong results!

in general $E(T(n)) \neq T(“E(n)”)$
Bubble Sort

BubbleSort (A: Array[1..n]) {
  for i from 1 to n do {
    for j from n downto i + 1 do {
    }
  }
}

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?
Mergesort

Basic Idea: divide and conquer

- **Divide** the problem into two (or more) subproblems:
  → split the array into two arrays of equal size

- **Conquer** the subproblems by solving them recursively:
  → sort both arrays using the sorting algorithm

- **Combine** the solutions of the subproblems:
  → 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

\[ T_{MS}(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
T_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T_{MS}\left(n - \left\lfloor \frac{n}{2} \right\rfloor \right) + cn & \text{if } n \geq 2
\end{cases} \]
Number of Comparisons in MergeSort (2)

Assume \( n = 2^k \), \( c \) constant:

\[
T_{MS}(2^k) = T_{MS}(2^{k-1}) + T_{MS}(2^{k-1}) + c \cdot 2^k
\]

\[
= 2T_{MS}(2^{k-1}) + 2^k c
\]

\[
= 2^2 T_{MS}(2^{k-2}) + 2 \cdot 2^{k-1} c + 2^k c
\]

\[
= \ldots
\]

\[
= 2^k T_{MS}(2^0) + 2^{k-1} \cdot 2^1 c + \ldots + 2^j \cdot 2^{k-j} c
\]

\[
\quad + \ldots + 2 \cdot 2^{k-1} c + 2^k c
\]

\[
= \sum_{j=1}^{k} 2^k c = ck \cdot 2^k = cn \log_2 n \in \Theta(n \log n)
\]
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.
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 ($A[q+1..r]$)

M. Bader: Fundamental Algorithms
Chapter 2: Sorting, Winter 2011/12
Partitioning the Array (Hoare’s Algorithm)

\[
\text{Partition } (A: \text{Array}[p..r]) : \text{Integer} \{ \\
\quad \text{// } x \text{ is the pivot:} \\
\quad x := A[p]; \\
\quad \text{// partitions grow towards each other} \\
\quad i := p; \; j := r; \; \text{// (partition boundaries)} \\
\quad \text{while true do } \{ \text{// } i<j: \text{ partitions haven’t met yet} \\
\quad \quad \text{// leave large elements in right partition} \\
\quad \quad \text{while } A[j] > x \text{ do } \{ j := j - 1; \}; \\
\quad \quad \text{// leave small elements in left partition} \\
\quad \quad \text{while } A[i] < x \text{ do } \{ i := i + 1; \}; \\
\quad \quad \text{// swap the two first ”wrong” elements} \\
\quad \quad \text{if } i < j \text{ then } \{ \\
\quad \quad \quad \text{exchange } A[i] \text{ and } A[j]; \\
\quad \quad \quad i := i + 1; \; j := j - 1; \\
\quad \quad \} \text{ else return } j; \\
\quad \}\}
\]
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 
    {
        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 in two halves:
  \[ T_{QS}^{\text{best}}(n) = 2T_{QS}^{\text{best}} \left( \frac{n}{2} \right) + \Theta(n) \]
- analogous to MergeSort:
  \[ T_{QS}^{\text{best}}(n) \in \Theta(n \log n) \]
Time Complexity of QuickSort

Best Case:
- assume that all partitions are split exactly in two halves:

\[ T_{QS}^{\text{best}}(n) = 2 T_{QS}^{\text{best}} \left( \frac{n}{2} \right) + \Theta(n) \]

- analogous to MergeSort:

\[ T_{QS}^{\text{best}}(n) \in \Theta(n \log n) \]

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

\[
T_{QS}^{\text{worst}}(n) = T_{QS}^{\text{worst}}(n-1) + T_{QS}^{\text{worst}}(1) + \Theta(n) \\
= T_{QS}^{\text{worst}}(n-1) + \Theta(n) = T_{QS}^{\text{worst}}(n-2) + \Theta(n-1) + \Theta(n) \\
= \ldots = \Theta(1) + \ldots + \Theta(n-1) + \Theta(n) \in \Theta(n^2)
\]
Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?

→ partition sizes always 1 and n-1
⇒ Θ(n^2)

- A is sorted in reverse order?
→ partition sizes always 1 and n-1
⇒ Θ(n^2)

- one partition has always at most \(a\) elements (for a fixed \(a\))?
→ same complexity as \(a = 1\) ⇒ Θ(n^2)

- partition sizes are always \(n(1-a)\) and \(na\) with \(0 < a < 1\)?
→ same complexity as best case ⇒ Θ(n \log n)

Questions:

- What happens in the “usual” case?
- Can we force the best case?
Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?
  → partition sizes always 1 and n-1 ⇒ Θ(n^2)

- A is sorted in reverse order?
  → partition sizes always 1 and n-1 ⇒ Θ(n^2)

- One partition has always at most \( a \) elements (for a fixed \( a \))?
  → same complexity as \( a = 1 \) ⇒ Θ(n^2)

- Partition sizes are always \( n(1-a) \) and \( na \) with \( 0 < a < 1 \)?
  → same complexity as best case ⇒ Θ(n \log n)
Time Complexity of QuickSort – Special Cases?

What happens if:

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

Questions:
- What happens in the “usual” case?
- Can we force the best case?
Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?
  → partition sizes always 1 and n-1 ⇒ $\Theta(n^2)$

- A is sorted in reverse order?
  → partition sizes always 1 and n-1 ⇒ $\Theta(n^2)$
Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?
  → partition sizes always 1 and n-1 ⇒ \( \Theta(n^2) \)

- A is sorted in reverse order?
  → partition sizes always 1 and n-1 ⇒ \( \Theta(n^2) \)

- one partition has always at most \( a \) elements (for a fixed \( a \))?
Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?
  → partition sizes always 1 and n-1 ⇒ $\Theta(n^2)$

- A is sorted in reverse order?
  → partition sizes always 1 and n-1 ⇒ $\Theta(n^2)$

- one partition has always at most $a$ elements (for a fixed $a$)?
  → same complexity as $a = 1$ ⇒ $\Theta(n^2)$
Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?
  → partition sizes always 1 and n-1 ⇒ Θ(n^2)

- A is sorted in reverse order?
  → partition sizes always 1 and n-1 ⇒ Θ(n^2)

- one partition has always at most a elements (for a fixed a)?
  → same complexity as a = 1 ⇒ Θ(n^2)

- partition sizes are always n(1 – a) and na with 0 < a < 1?
Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?
  → partition sizes always 1 and n-1 ⇒ Θ(n²)

- A is sorted in reverse order?
  → partition sizes always 1 and n-1 ⇒ Θ(n²)

- one partition has always at most a elements (for a fixed a)?
  → same complexity as a = 1 ⇒ Θ(n²)

- partition sizes are always n(1 – a) and na with 0 < a < 1?
  → same complexity as best case ⇒ Θ(n log n)
Time Complexity of QuickSort – Special Cases?

What happens if:

- A is already sorted?
  → partition sizes always 1 and n-1 ⇒ \( \Theta(n^2) \)

- A is sorted in reverse order?
  → partition sizes always 1 and n-1 ⇒ \( \Theta(n^2) \)

- one partition has always at most \( a \) elements (for a fixed \( a \))?
  → same complexity as \( a = 1 \) ⇒ \( \Theta(n^2) \)

- partition sizes are always \( n(1 - a) \) and \( na \) with \( 0 < a < 1 \)?
  → same complexity as best case ⇒ \( \Theta(n \log n) \)

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 {
        q := RandPartition (A);
        RandQuickSort (A[p..q]);
        RandQuickSort (A[q+1..r]);
    }
}
Time Complexity of RandQuickSort

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

\[
\bar{T}_{RQS}(n) = \sum_{j=0}^{n-1} P(k = j) T_{RQS}(n|k=j) + \Theta(n)
\]

\[
= \frac{1}{n} \left( (T_{RQS}(1) + T_{RQS}(n-1)) + \sum_{j=1}^{n-1} (T_{RQS}(j) + T_{RQS}(n-j)) \right) + \Theta(n)
\]
Average Case Complexity of RandQuickSort (2)

The expected runtime of RandQuickSort is

\[
\overline{T}_{\text{RQS}}(n) = \sum_{j=0}^{n-1} P(k = j) T_{\text{RQS}}(n|k=j) + \Theta(n)
\]

\[
= \frac{1}{n} \left( (T_{\text{RQS}}(1) + T_{\text{RQS}}(n-1)) + \sum_{j=1}^{n-1} (T_{\text{RQS}}(j) + T_{\text{RQS}}(n-j)) \right) + \Theta(n)
\]

- Note: all \( T_{\text{RQS}}(j) \) depend on the choice of the next pivot elements
- simplifying assumption: \( T_{\text{RQS}}(j) = \overline{T}_{\text{RQS}}(j) \)
Average Case Complexity of RandQuickSort (3)

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

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

Thus

$$\bar{T}_{RQS}(n) = \frac{1}{n} \left( \sum_{j=1}^{n-1} \left( \bar{T}_{RQS}(j) + \bar{T}_{RQS}(n - j) \right) \right) + \Theta(n)$$

$$= \frac{2}{n} \sum_{j=1}^{n-1} \bar{T}_{RQS}(j) + \Theta(n)$$
Average Case Complexity of RandQuickSort (3)

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

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

Thus

$$\bar{T}_{RQS}(n) = \frac{1}{n} \left( \sum_{j=1}^{n-1} (\bar{T}_{RQS}(j) + \bar{T}_{RQS}(n - j)) \right) + \Theta(n)$$

$$= \frac{2}{n} \sum_{j=1}^{n-1} \bar{T}_{RQS}(j) + \Theta(n)$$

Solve recurrence equation to obtain:

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