2.3 General Amortisation Analysis

The use of amortised complexity as elaborated for unbounded arrays can in fact be generalised to arbitrary data structures

Consider an arbitrary data structure, which according to our initial motivation, is just a realisation of an ADT

We can therefore assume that there is a set of states $\mathcal{S}$ defined by the data structure, there are **operations** $op \in \mathcal{O}$ that define **state transitions**, and each state $S \in \mathcal{S}$ is determined by a collection of **size parameters** $p_1, \dots, p_k$

A transition from state S to state S' by means of an operation op has a **cost** $T_{op}(S)$

Then we can assume sequences $S_0 \stackrel{op_1}{\rightarrow} S_1 \stackrel{op_2}{\rightarrow} S_2 \stackrel{op_3}{\rightarrow} \cdots \stackrel{op_n}{\rightarrow} S_n$

If F is such a sequence, then the cost associated with F is $T(F) = \sum_{i=1}^{n} T_{op_i}(S_{i-1})$

Amortised Time Bounds

Now consider families $\{A_{op} \mid op \in \mathcal{O}\}$ of functions $A_{op} : \mathcal{S} \rightarrow \mathbb{R}_{\geq 0}$

We want to have "easier" functions A_{op} that can be used instead of the actual cost functions T_{op}, i.e. they define appropriate time bounds

That is, for any sequence F of states as above, we request that (up to some additive constant) the total actual execution time, i.e. $T(F)$, is bounded by the total amortised execution time

A family $\{A_{op} : \mathcal{S} \rightarrow \mathbb{R}_{\geq 0} \mid op \in \mathcal{O}\}$ is called a **family of amortised time bounds** iff there exists a constant $c \geq 0$ such that $T(F) \leq c + A(F)$ holds for all sequences of states in $\mathcal{S}$ with state transitions defined by the operations $op \in \mathcal{O}$, where

$$A(F) = c + \sum_{i=1}^{n} A_{op_i}(S_{i-1})$$

and the constant $c \geq 0$ does not depend on F

Clearly, $A_{op} = T_{op}$ satisfies the required condition for a family of amortised time bounds, but it is far from being "simple"

Potential Functions

The **potential method** for amortisation analysis formalises the technique we used for the unbounded arrays

For this we use a **potential function** $pot : \mathcal{S} \to \mathbb{R}_{\geq 0}$, which assigns to each state a non-negative **potential** or **balance**

We then define the **amortised cost** of operation op in state S by

$$A_{op}(S) = pot(S') - pot(S) + T_{op}(S) \ ,$$

where S' is the state resulting from the execution of op in state S

next state

Theorem. Let $pot : \mathcal{S} \to \mathbb{R}_{\geq 0}$ be a potential function on the set $\mathcal{S}$ of states of a data structure. For an operation $op \in \mathcal{O}$ of the data structure with $S \xrightarrow{op} S'$ define a function

$$A_{op} : \mathcal{S} \to \mathbb{R}_{\geq 0} \qquad \text{by} \qquad A_{op}(S) = pot(S') - pot(S) + T_{op}(S) \ .$$

Then $\{A_{op} \mid op \in \mathcal{O}\}$ is a family of amortised time bounds.

Proof

Consider an arbitrary sequence F defined as $S_0 \xrightarrow{op_1} S_1 \xrightarrow{op_2} S_2 \xrightarrow{op_3} \cdots \xrightarrow{op_n} S_n$

Then we get

$$\begin{aligned}
A(F) \quad &= \quad \sum_{i=1}^{n} A_{op_i}(S_{i-1}) \quad = \quad \sum_{i=1}^{n} pot(S_i) - pot(S_{i-1}) + T_{op_i}(S_{i-1}) \\
&= \quad \boxed{pot(S_n)} - pot(S_0) + \sum_{i=1}^{n} T_{op_i}(S_{i-1}) \\
&\geq \quad \sum_{i=1}^{n} T_{op_i}(S_{i-1}) - pot(S_0) \quad = \quad T(F) - pot(S_0)
\end{aligned}$$

Some positive number

With $c = pot(S_0) \geq 0$ we get $T(F) \leq c + A(F)$ as claimed

Example

Let us review the case of an unbounded array, where the size parameters of a state are the size n of the representing array, and the size m of list, i.e. $m \leq n$

For simplicity without loss of generality let the cost of a copy operation be just 1

Define $pot(n, m) = \max(3m - n, n/2)$, i.e. it is $n/2$ for $m \leq n/2$ and it is $3m - n$ for $m \geq n/2$

The operations *insert* and *append* increase m by 1, so the difference $pot(S') - pot(S)$ is either 0 or 3

For *append* the actual cost is 1, so the amortised cost is either 1 or 4, both bounded by $4 \in O(1)$

For *insert* the actual cost is m, so the amortised cost is either $m + \overset{0}{1}$ or $m + 3$, both in $O(m)$

The operation *delete* decreases m by 1, so the difference $pot(S') - pot(S)$ is either 0 or -3

Example / cont.

For *delete* the actual cost is m, so the amortised cost is either m or $m-3$, both in $O(m)$

The operation *allocate* in only applicable, when $m = n$ holds, in which case doubles n, which gives

$$pot(S') - pot(S) = \max(3m-2n, n) - \max(3m-n, n/2) = n - 2n = -n$$

For *allocate* the actual cost is n, so the amortised cost is 0

The operation *deallocate* in only applicable, when $m = n/4$ holds, in which case halves n, which gives

$$pot(S') - pot(S) = \max(3m-n/2, n/4) - \max(3m-n, n/2) = n/4 - n/2 = -n/4$$

For *deallocate* the actual cost is $n/\cancel{4}$, so the amortised cost is 0

Universality of the Potential Method

Theorem. Let $\{B_{op} : \mathcal{S} \to \mathbb{R}_{\geq 0} \mid op \in \mathcal{O}\}$ be a family of amortised time bounds. Then there exists a potential function $pot : \mathcal{S} \to \mathbb{R}_{\geq 0}$ such that $A_{op}(S) \leq B_{op}(S)$ holds for all states $S \in \mathcal{S}$ and all operations $op \in \mathcal{O}$, where the functions A_{op} are defined by

$$A_{op}(S) = pot(S') - pot(S) + T_{op}(S) \; .$$

In other words, the potential method is strong enough to obtain any family of amortised time bounds

Proof. According to the definition of amortised time bounds there exists a constant $c \geq 0$ such that $T(F) \leq c + B(F)$ holds for any sequence F of operations

Define the potential function *pot* by

$$pot(S) = \underbrace{\inf}_{\text{最小下界}}\{B(F) + c - T(F) \mid F \text{ is a sequence of operations with final state } S\}$$

As $B(F) + c - T(F) \geq 0$ holds, we have $pot(S) \geq 0$ for all states $S \in \mathcal{S}$, so *pot* is a potential function

Proof (cont.)

Our previous theorem shows that $\{A_{op} : \mathcal{S} \to \mathbb{R}_{\geq 0} \mid op \in \mathcal{O}\}$ is a family of amortised time bounds

It remains to show that $A_{op}(S) \leq B_{op}(S)$ holds for all states $S \in \mathcal{S}$ and all operations $op \in \mathcal{O}$; it suffices to show $A_{op}(S) \leq B_{op}(S) + \varepsilon$ for arbitrary $\varepsilon > 0$

For arbitrary $\varepsilon > 0$ there exists a sequence F of operations with final state S such that $B(F) + c - T(F) \leq pot(S) + \varepsilon$

We extend it by $op \in O$ to a sequence F'; then for the final state S' of F' the definition of *pot* implies $pot(S') \leq B(F') + c - T(F')$

As $B(F') = B(F) + B_{op}(S)$ and $T(F') = T(F) = T_{op}(S)$ hold, we obtain

$$\begin{aligned} A_{op}(S) &= pot(S') - pot(S) + T_{op}(S) \\ &\leq (B(F') + c - T(F')) - (B(F) + c - T(F) - \varepsilon) + T_{op}(S) \\ &= (B(F') - B(F)) - (T(F') - T(F) - T_{op}(S)) + \varepsilon \\ &= B_{op}(S) + \varepsilon \end{aligned}$$

2.4 Sorting Algorithms

So far we ignored the *sort*() operation associated with lists

For sorting there exist many algorithms, which roughly fall into two classes:

Elementary Sorting Algorithms. bubblesort (already known), selection sort, insertion sort, comb sort

We will introduce and analyse insertion and selection sort

Fast Sorting Algorithms. mergesort, quicksort, radixsort, heapsort, shellsort, countingsort

We will introduce and analyse mergesort and quicksort

heapsort will be discussed in the section of the course dealing with heap data structures

More sorting algorithms are left for specific courses on algorithms

Selection Sort

The idea of selection sort is rather simple: *select* and remove the smallest element of the input list and append it to the output list

That is, using *outlist* initialised to [] we just have to iterate the rule:

```
IF     $inlist \neq []$
THEN   CHOOSE  $x \in inlist$ WITH $\forall y . y \in inlist \rightarrow x \leq y$
       DO      PAR
                   $inlist := inlist - [x]$
                   $outlist := outlist + [x]$
               ENDPAR
       ENDDO
ENDIF
```

Clearly, selection sort has time complexity in $O(n^2)$, where n is the length of the list to be sorted—we have to search the whole *inlist* to find a minimal element

Example

selected item	inlist	outlist
	[8, 13, 5, 1, 2, 4, 3, 7, 6]	[]
1	[8, 13, 5, 2, 4, 3, 7, 6]	[1]
2	[8, 13, 5, 4, 3, 7, 6]	[1, 2]
3	[8, 13, 5, 4, 7, 6]	[1, 2, 3]
4	[8, 13, 5, 7, 6]	[1, 2, 3, 4]
5	[8, 13, 7, 6]	[1, 2, 3, 4, 5]
6	[8, 13, 7]	[1, 2, 3, 4, 5, 6]
7	[8, 13]	[1, 2, 3, 4, 5, 6, 7]
8	[13]	[1, 2, 3, 4, 5, 6, 7, 8]
13	[]	[1, 2, 3, 4, 5, 6, 7, 8, 13]

Insertion Sort

Insertion sort is somehow dual to selection sort: remove an arbitrary element from the input list and insert it into the output list according to the order

That is, using *outlist* initialised to [] we just have to iterate the rule:

```
IF      $inlist \neq []$
THEN    LET $x = first(inlist)$
        IN  PAR
                $inlist := \mathbf{I}\ell.[x] + \ell = inlist$
                LET  $\ell_1 = \mathbf{I}\ell.sublist(\ell, outlist) \wedge \forall y.y \in \ell \leftrightarrow y < x$,
                     $\ell_2 = \mathbf{I}\ell.outlist = \ell_1 + \ell$
                IN $outlist := \ell_1 + [x] + \ell_2$
                ENDLET
            ENDPAR
        ENDLET
ENDIF
```

Clearly, insertion sort has time complexity in $O(n^2)$, where n is the length of the list to be sorted— for every element we have to search the whole *outlist* to find the correct position for insertion

Example

selected item	inlist	outlist
	[8, 13, 5, 1, 2, 4, 3, 7, 6]	[]
8	[13, 5, 1, 2, 4, 3, 7, 6]	[8]
13	[5, 1, 2, 4, 3, 7, 6]	[8, 13]
5	[1, 2, 4, 3, 7, 6]	[5, 8, 13]
1	[2, 4, 3, 7, 6]	[1, 5, 8, 13]
2	[4, 3, 7, 6]	[1, 2, 5, 8, 13]
4	[3, 7, 6]	[1, 2, 4, 5, 8, 13]
3	[7, 6]	[1, 2, 3, 4, 5, 8, 13]
7	[6]	[1, 2, 3, 4, 5, 7, 8, 13]
6	[]	[1, 2, 3, 4, 5, 6, 7, 8, 13]

Mergesort

Recall the ***mergesort*** algorithm that has been introduced as an example for recursion in **Discrete Mathematics**

The algorithm applies the **divide & conquer** strategy: it splits a list into sublists of (almost) equal length and sorts each sublist recursively

The sorting of the sublists can be done by *mergesort* as long as the lists are long enough; for short lists a basic sorting algorithm (selection, insertion or bubble sort) is used

For the separation of “long enough” and “short” lists we use a threshold *thr* (we will discuss appropate values for *thr* later)

When sorted sublists are returned, they are merged by a recursive *merge* algorithm, for which it suffices to scan the lists from the beginning to the end

Mergesort / cont.

```
sorted_list ← $sort$(unsorted_list) =
  IF sorted_list = $undef$
  THEN LET $n$ = length(unsorted_list)
  IN PAR
    IF $n \leq thr$ THEN sorted_list ← $basic\_sort$(unsorted_list)ENDIF
    IF $n > thr \wedge$ sorted_list$_1$ = $undef \wedge$ sorted_list$_2$ = $undef$
    THEN LET list$_1$ = $\mathbf{I}L$.length$(L) = \lfloor \frac{n}{2} \rfloor \wedge \exists L'$.concat$(L, L')$ = unsorted_list
            IN    LET list$_2$ = $\mathbf{I}L'$.concat(list$_1$, $L'$) = unsorted_list
                  IN  PAR  sorted_list$_1$ ← $sort$(list$_1$)
                           sorted_list$_2$ ← $sort$(list$_2$)
                      ENDPAR
    ENDIF
    IF $n > thr \wedge$ sorted_list$_1 \neq undef \wedge$ sorted_list$_2 \neq undef$
    THEN sorted_list ← $merge$(sorted_list$_1$,sorted_list$_2$) ENDIF
    ENDPAR
  ENDIF
```

Mergesort / cont.

```
merged_list ← *merge*(inlist$_1$,inlist$_2$) =
    IF merged_list = *undef* THEN PAR
    IF inlist$_1$ = [] THEN merged_list := inlist$_2$ ENDIF
    IF inlist$_2$ = [] THEN merged_list := inlist$_1$ ENDIF
    IF inlist$_1 \neq [] \wedge$ inlist$_2 \neq []$
    THEN    LET $x_1$ = head(inlist$_1$), restlist$_1$ = tail(inlist$_1$),
            $x_2$ = head(inlist$_2$), restlist$_2$ = tail(inlist$_2$) IN PAR
        IF $x_1 \leq x_2 \wedge$ merged_restlist = *undef*
        THEN merged_restlist ← *merge*(restlist$_1$, inlist$_2$) ENDIF
        IF $x_1 > x_2 \wedge$ merged_restlist = *undef*
        THEN merged_restlist ← *merge*(inlist$_1$, restlist$_2$) ENDIF
        IF $x_1 \leq x_2 \wedge$ merged_restlist $\neq$ *undef*
        THEN merged_list := concat([$x_1$],merged_restlist)) ENDIF
        IF $x_1 > x_2 \wedge$ merged_restlist $\neq$ *undef*
        THEN merged_list := concat([$x_2$],merged_restlist) ENDIF
      ENDPAR
    ENDIF ENDPAR
```

Example

Lists	**lengths**
[8, 13, 5, 1, 2, 4, 3, 7, 6]	9
[8, 13, 5, 1], [2, 4, 3, 7, 6]	4+5
[8, 13], [5, 1], [2, 4], [3, 7, 6]	2+2+2+3
[8], [13], [5], [1], [2], [4], [3], [7, 6]	1+1+1+1+1+1+1+2
$\underbrace{[8], [13]}, \underbrace{[5], [1]}, \underbrace{[2], [4]}, [3], \underbrace{[7], [6]}$	1+1+1+1+1+1+1+1+1 (sorted)
$\underbrace{[8, 13], [1, 5]}, [2, 4], \underbrace{[3], [6, 7]}$	2+2+2+1+2 (sorted)
$[1, 5, 8, 13], \underbrace{[2, 4], [3, 6, 7]}$	4+2+3 (sorted)
$\underbrace{[1, 5, 8, 13], [2, 3, 4, 6, 7]}$	4+5 (sorted)
[1, 2, 3, 4, 5, 6, 7, 8, 13]	9 (sorted)

The baces show the lists that are merged

Example (merge)

selected element	**restlist$_1$**	**restlist$_2$**	**merged_list**
	[1, 5, 8, 13]	[2, 3, 4, 6, 7]	[]
1	[5, 8, 13]	[2, 3, 4, 6, 7]	[1]
2	[5, 8, 13]	[3, 4, 6, 7]	[1, 2]
3	[5, 8, 13]	[4, 6, 7]	[1, 2, 3]
4	[5, 8, 13]	[6, 7]	[1, 2, 3, 4]
5	[8, 13]	[6, 7]	[1, 2, 3, 4, 5]
6	[8, 13]	[7]	[1, 2, 3, 4, 5, 6]
7	[8, 13]	[]	[1, 2, 3, 4, 5, 6, 7]
	[]	[]	[1, 2, 3, 4, 5, 6, 7, 8, 13]

Complexity Analysis

For the *merge* algorithm we can proceed as before to see that its complexity $g(n)$ is in $\mathbf{\Theta}(n)$

For the *mergesort* algorithm we then see that its complexity $f(n)$ satisfies

$$f(n) \;=\; \begin{cases} a_1 \cdot n^2 + b_1 \cdot n + c & \text{if } n \leq thr \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + g(n) + d & \text{else} \end{cases}$$

with some constants a_1, a_2, c, d

Replacing n by 2^n this leads to a linear recurrence equation

$$h_n - 2h_{n-1} = g(2^n) + d = a_2 2^n + b_2 + d$$

with some constants a_2, b_2

Complexity Analysis / cont.

The characteristic polynomial is $(x-2)^2(x-1)$, which gives rise to

$$h_n = c_1 2^n + c_2 n 2^n + c_3$$

and further

$$f(n) \;=\; c_1 \cdot n + c_2 n \log n + c_3 \in O(n \log n \mid \varphi(n)) \ ,$$

where the condition $\varphi(n)$ expresses that n is a power of 2

As $n \log n$ is smooth and $f(n)$ is almost everywhere monotone increasing, we can infer $f(n) \in O(n \log n)$

Actually, we have shown $f(n) \in \mathbf{\Theta}(n \log n)$

Number of Comparisons

Mergesort exploits the total order on the set T the elements of the list are in

Thus, the comparison of two list elements with respect to the order is decisive for sorting

We therefore look into an additional complexity measure not counting the number of elementary operations, but the number of comparisons instead

We will show the following:

- The *merge* algorithm on two sorted lists ℓ_1 and ℓ_2 with $|\ell_1| + |\ell_2| = n$ requires $n - 1$ comparisons
- The *mergesort* algorithm on a list ℓ of length n requires at ~~most~~ least $\lceil n \log_2 n \rceil$ comparisons (if the threshold *thr* is 1)

Using a different threshold value has only a marginal effect on the number of comparisons

Proof

In the *merge* algorithm, whenever an element x_1 is appended to the output list, this is done after a comparison $x_1 \leq x_2$, except for the last element

Consequently, for n elements we have $n - 1$ such comparisons, which shows the first part of our claim

For *mergesort* let $k(n)$ be the number of comparisons needed for an input list of length n

Then we get $k(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ k(\lfloor \frac{n}{2} \rfloor) + k(\lceil \frac{n}{2} \rceil) + n - 1 & \text{else} \end{cases}$

The two summands $k(\lfloor \frac{n}{2} \rfloor)$ and $k(\lceil \frac{n}{2} \rceil)$ stem from the two recursive calls of *mergesort*, and the summand $n - 1$ from the final *merge*

Proof / cont.

Substituting n by 2^n we obtain a linear recurrence equation $d_n - 2d_{n-1} = 2^n - 1$ with the characteristic polynomial $(x-2)^2(x-1)$

So the general solution takes the form $d_n = c_1 \cdot 2^n + c_2 \cdot n \cdot 2^n + c_3$ with initial condition $d_0 = 0$, $d_1 = 1$ and $d_2 = 5$

The initial conditions define a system of linear equations with the unique solution $c_2 = c_3 = 1$ and $c_1 = -1$, i.e. $d_n = (n-1) \cdot 2^n + 1$

Resubstitution gives $k(n) = (\log_2 n - 1) \cdot n + 1 = n \cdot \log_2 n - (n-1) \leq \lceil n \cdot \log_2 n \rceil$

This extends to all n, not only powers of 2, as the functions are smooth, which shows the second claim

A Lower Bound

We will now sketch a proof of a lower bound for comparison-based sorting algorithms

Theorem. Every comparison-based sorting algorithm requires $n \log_2 n - c \cdot n$ comparisons with some constant c.

Proof. Let the list to be sorted contain the elements $x_1, \dots, x_n$

Define a binary decision tree with non-leaf vertices labelled by a comparison $x_i \leq x_j$, and leaves labelled by permutations $\sigma \in S_n$, i.e. a leaf corresponds to a possible sorting $x_{\sigma(1)}, \dots, x_{\sigma(n)}$

If $v_0, \dots, v_d$ is a path from the root to a leaf, the label of v_k is $x_{k_i} \leq x_{k_j}$ (for $0 \leq k \leq d-1$) and the label of v_d is σ, then

- v_k corresponds to $\varphi_k = \begin{cases} x_{k_i} \leq x_{k_j} & \text{if } v_{k+1} \text{ is the left successor of } v_k \\ x_{k_j} < x_{k_i} & \text{if } v_{k+1} \text{ is the right successor of } v_k \end{cases}$
- $\{\varphi_0, \dots, \varphi_{d-1}\}$ imply $x_{\sigma(1)} \leq \dots \leq x_{\sigma(n)}$

Proof / cont.

If the tree is minimal, it has exactly $n!$ leaves (as $n! = |S_n|$), and the depth d of the tree is the number of required comparisons

As a binary tree of depth d can have at most 2^d leaves, we must have $2^d \geq n!$

Apply the **Stirling formula**: $n! = (1 + \gamma_n)\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ with $\gamma_n \in O(1/n)$, in particular $n! \geq \left(\frac{n}{e}\right)^n$ (here e is the Euler number)

This gives

$$d \geq \log_2 n! \geq \log_2 \left(\frac{n}{e}\right)^n = n \cdot (\log_2 n - \log_2 e) = n \cdot \log_2 n - n \cdot \log_2 e$$

which completes the proof

Further Remarks

We note without proof that the theorem can be generalised to the average number of comparisons required by a sorting algorithm

Theorem. Every comparison-based sorting algorithm requires on average $n \log_2 n - c \cdot n$ comparisons with some constant c.

With these theorems we can conclude that the best worst case and average case time complexity for sorting algorithms we can expect is in $O(n \cdot \log n)$

Then up to some constants *mergesort* is an optimal sorting algorithm

Concerning the threshold *thr* used in the algorithm we can use the exact solutions of the recurrence equation for *mergesort* and compare it with the exact number of steps required for a basic sorting algorithm such as bubble sort, insertion sort or selection sort

Choose a value for *thr* such that for $n \leq thr$ the basic quadratic sorting algorithm performs better than *mergesort*—the value should be between 60 and 70

Further Remarks on mergesort

The above implementation suffers from repeated search through linked list to find the address of the node in the middle

This can be optimised by calculating indices in advance and using splicing to isolate small-sized sublists (without dummy node) and sorting them in-place, so that only *merge* has to be applied

The *mergesort* algorithm can also be based on unbounded arrays

Based on arrays the access to the middle element of a list is an operation in $O(1)$

However, when arrays are used, it is not possible to realise an in-place implementation of *merge*

These differences become particularly relevant when dealing with large lists that require the use of external secondary storage

Quicksort

Another (fast) alternative for sorting a list is the ***quicksort*** algorithm, which was has also been introduced in **Discrete Mathematics**

The algorithm chooses an arbitrary element x from the list (usually the first), then splits the list into two sublists

Different to mergesort the sublists are built by comparison of elements with x

Then no merging is necessary—the sorted sublists are simply concatenated

One disadavantage is that the sublist may have very different lengths

Quicksort / cont.

```
$\text{sorted\_list} \leftarrow \mathit{qsort}(\text{unsorted\_list}) =$
    PAR IF $\text{unsorted\_list} = [\,]$ THEN $\text{sorted\_list} := [\,])$ ENDIF
        IF $\text{unsorted\_list} \neq [\,] \wedge \text{sorted\_list}_1 = \mathit{undef} \wedge \text{sorted\_list}_2 = \mathit{undef}$
        THEN LET $x = \text{head}(\text{unsorted\_list})$ IN
             LET $\text{unsorted\_list}_1 = \text{sublist}[y \mid y < x](\text{unsorted\_list})$ IN
             LET $\text{unsorted\_list}_2 = \text{sublist}[y \mid y > x](\text{unsorted\_list})$ IN
             PAR $\text{sorted\_list}_1 \leftarrow \mathit{qsort}(\text{unsorted\_list}_1)$
                 $\text{sorted\_list}_2 \leftarrow \mathit{qsort}(\text{unsorted\_list}_2)))$
             ENDPAR
        ENDIF
        IF $\text{unsorted\_list} \neq [\,] \wedge \text{sorted\_list}_1 \neq \mathit{undef} \wedge \text{sorted\_list}_2 \neq \mathit{undef}$
        THEN $\text{sorted\_list} := \text{concat}(\text{sorted\_list}_1,$
                 $\text{concat}(\text{head}(\text{unsorted\_list}), \text{sorted\_list}_2))$
        ENDIF
    ENDPAR
```

Complexity Analysis

We will look at the time complexity of quicksort in the worst case and on average

First, let $C(n)$ denote the number of comparisons in the worst case for any choice of *pivot*, where n is the length of the input list

The three sublists are built by comparing each list element with the *pivot* element, which makes $n-1$ comparisons

Assume there are k elements in $list1$, i.e. smaller than *pivot*, and ℓ elements in $list2$, i.e. larger than *pivot*

Then we have $C(0) = C(1) = 0$ and

$$C(n) \leq n - 1 + \max\{C(k) + C(\ell) \mid 0 \leq k \leq n-1 \wedge 0 \leq \ell < n - k\}$$

Using induction we show easily $C(n) \leq \dfrac{n(n-1)}{2}$

Complexity Analysis / cont.

The worst case occurs, when all list elements are different and always the smallest or largest element is taken as *pivot*

In this case we have $C(n) = n - 1 + C(n-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$

On average, however, the complexity is much better

Theorem. On average, the number of comparisons required by *quicksort* is

$$\bar{C}(n) \leq 2 \cdot n \cdot \log n = (2 \log 2) \cdot n \cdot \log_2 n \leq 1.45 \cdot n \cdot \log_2 n \ .$$

Proof. Let the list elements be $x_1, \dots, x_n$ with $x_1 \leq \cdots \leq x_n$

Then x_i and x_j are compared at most once, and only, if one of them was picked as *pivot* element

Proof / cont.

Define $\delta_{i,j} = \begin{cases} 1 & \text{if } x_i \text{ and } x_j \text{ are compared} \\ 0 & \text{else} \end{cases}$

Then $\delta_{i,j}$ is a Bernoulli-distributed random variable—let $p_{i,j}$ be the probability $P(\delta_{i,j} = 1)$

Then we have

$$\bar{C}(n) = E\left(\sum_{i=1}^{n} \sum_{j=i+1}^{n} \delta_{i,j}\right) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E(\delta_{i,j}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j}$$

Proof / cont.

Claim. We have $p_{i,j} = \dfrac{2}{j-i+1}$ for $1 \leq i < j \leq n$.

Proof of the claim. As long as *pivot* is not taken from $M = \{x_i, \dots, x_j\}$, x_i and x_j aree not compared, and all elements of M are in the same sublist

The probability to select *pivot* from M is $\dfrac{1}{|M|} = \dfrac{1}{j-i+1}$

If this selection is x_i or x_j we obtain $\delta_{i,j} = 1$; otherwise, x_i and x_j will never be compared, i.e. $\delta_{i,j} = 0$

This implies $p_{i,j} = \dfrac{2}{j-i+1}$ as claimed

Proof / cont.

Now compute

$$\begin{aligned}
\bar{C}(n) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j} &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} &\leq \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} \\
&= 2n \sum_{k=2}^{n} \frac{1}{k} &\leq 2n \int_{1}^{n} \frac{1}{x} dx \\
&= 2n \log n \ ,
\end{aligned}$$

which completes the proof

Further Improved Sorting

We now look at a sorting algorithm with a complexity better than $\Theta(n \log n)$

Of course, this is not possible without further assumption on the set T of elements in a list

Without such assumptions an algorithm can only compare elements of T using the total order $\leq$, and then we have seen a complexity in $\Theta(n \log n)$ is optimal

This changes for instances for $T = \mathbb{N}$, because then we can exploit arithmetic operators on T

More generally, assume that we have a function $\mathit{key} : T \to \mathbb{N}$ so that we can define a partial order on T by $t_1 \leq t_2$ iff $\mathit{key}(t_1) \leq \mathit{key}(t_2)$ (and $t_1 < t_2$ iff $\mathit{key}(t_1) < \mathit{key}(t_2)$)

With this partial order relax the definition of "sorted": a list with elements in T is ***sorted*** iff for all $i \leq j$ we have $\mathit{key}(\mathit{List}(i)) \leq \mathit{key}(\mathit{List}(j))$

In other words: elements of T with the same *key* value can appear in a sorted list in any order

KSort

Let us first assume that we have a small number $K \in \mathbb{N}$ such that $key(t) \in \{0, \dots, K-1\}$ holds for all $t \in T$ appearing in a list ℓ

The idea of the **KSort** algorithm is quite simple:

- Scan the list from the first to the last element
- When the i'th list element has key k, put it into the k'th **bucket**, i.e. append it to a list $list_k$ (these lists are initially empty)
- Finally, concatenate the lists $list_0, \dots, list_{K-1}$ to obtain the sorted list

If the list has length n, we have to execute n append operations plus finally $K-1$ concatenate operations, so the time complexity of KSort is in $O(n+K)$ (provided we exploit linked lists)

Radix Sort

The **radix sort** algorithm generalises KSort to arbitrary key values in $\mathbb{N}$

Assume that we use a K-ary representation of the key values (i.e. we need digits $0, \dots, K-1$), and that we need to represent key values in the range $0, \dots, K^d - 1$

Then apply the KSort algorithm d times, first for the least significant digits, then the second least significant digit, etc. finishing with KSort for the most significant digit

As we apply KSort d times, each time with K buckets, we obtain time complexity in $O(d(n + K))$

However, it is still necessary to see that the result of the radix sort algorithm is indeed a sorted list

Correctness of Radix Sort

First notice the following obvious property of the KSort algorithm:

If $key(List(i)) = key(List(j))$ holds for $i < j$, then after sorting with KSort $List(i)$ will appear before $List(j)$ in the sorted list

This is a consequence of KSort scanning the input list from the first to the last element and appending each list element to the appropriate bucket, i.e. the order of list elements with the same key will not be changed

Consequently, if the i'th digit of two keys is the same, in the i'th round of radix sort (using KSort with the K buckets for the i'th digit) the previous sorting on grounds of the $(i - 1)$'th digit will not be changed

So the result of radix sort is sorted by the most important digit used in the last round; elements with a key with the same most important digit are sorted according to the second most important digit, etc.

This shows that the result of radix sort is a sorted list

2.5 Remarks on Secondary Storage

We now discuss briefly lists of length that exceeds the capacity of main memory

The units of secondary storage are blocks (files are sequences of such blocks) and each block is linearly organised (same as main memory)

So addresses of words in secondary storage are given by the block identifier and an offset into the block

When dealing with external memory, some blocks are buffered in main memory, and access is only possible, after a block has been moved to the buffer

That is, addresses in blocks can be translated to buffer addresses (this is part of main memory)—this is managed by a buffer manager

So the main challenge is to minimise **paging**, i.e. the replacement of blocks in the buffer

Lists in Secondary Storage

For most operations it is advantageous, if contiguous elements of a list are kept together in the same block

This applies to both the representation by "unbounded" arrays, as well as linked list

However, as allocation and deallocation operations always create new lists, it is usually a better idea to exploit linked lists when dealing with secondary storage

We will later discuss tree-based index structures that compensate for the increased diffulty to locate individual list elements

Sorting operations reorganise lists, so if secondary storage is used, it is advisable to exploit sorting algorithms that can be implemented in-place and do not require a lot of paging

We will therefore concentrate on mergesort in our discussion here

External Sorting

If we consider the key idea of mergesort again, we first split the given list into two sublists of almost equal size

We may as well consider a split into k sublists for an arbitrary number k

If the given list resides in external storage, a natural split is already given by the blocks containing the list elements

Therefore, external sorting with mergesort can first sort the sublists in all blocks separately (also in parallel)

For this it is sufficient to load a single block into main memory and apply an arbitrary sorting algorithm (not necessarily mergesort) to the list stored in the block, and write the block back to external storage

External Sorting / 2

If we can assume that the list elements in each block are already sorted, the remaining task is the merging of sorted blocks

For merging it suffices to load two blocks into main memory and use a third block for the output

For (sorted) sublists spanning more than one block, follow-on blocks are only needed in memory, once all elements in the previous block have already been moved to the outlist

If the block for the outlist is filled, it can be written back to secondary storage, while the sorting continues with a new block

We omit further details here

2.6 Stacks, Queues and Deques

We now look into three simpler sequence data structures that differ from lists by the reduced set of operations:

- **Stacks** (also called Last-in-First-out (LIFO) sequences) are finite (possibly empty) sequences $s(0), \dots, s(n)$ that can only be updated by adding a new element $s(n+1)$ (**push**) or removing $s(n)$ from the stack (**pop**)

  To emphasise the intuitive understand of the word "stack" we call $s(n)$ the (**top**) element of the stack are finite (possibly empty) sequences $s(0), \dots, s(n)$, where the only updates are to push an element at the end, i.e. add a new element $s(n+1)$, or pop an element at the front, i.e. remove $s(0)$

- **Queues** (First-in-First-out (FIFO) sequences)

- **Deques** ("double-ended queues") permit push and pop updates at the front and the end of the sequence

We will only look at representations by arrays, leaving out considerations on linked (pointer-based) structures

Stack Operations

In more detail we consider the following (most important) operations on **stacks**:

- ***isEmpty*** returns **true**, if the stack contains no element, and **false** otherwise—the complexity is clearly in $\Theta(1)$
- ***pop*** removes and returns the top element of a non-empty stack; it is undefined for empty stacks—the (amortised) complexity is clearly in $\Theta(1)$
- ***top*** returns the top element of a non-empty stack without changing the stack; it is undefined for empty stacks—the complexity is clearly in $\Theta(1)$
- ***push*** adds a new top element to the stack—the (amortised) complexity is clearly in $\Theta(1)$

Same as for lists with direct access we have to consider amortised complexity, as a representing array may needed to be enlarged in case of a *push* (analogous to *append* on lists), and shrunk in case of a *pop* (analogous to *delete* on lists)

Representation of Queues

For **queues** it is advantageous to consider a representation by "circular" arrays:

- Instead of shifting elements when the front element of the queue is removed we simply keep the array index of the first element in addition to the index of the last element

- Then $last - first + 1$ is the length of the queue

- A new element can still be added to the array, as long as the length of the queue does not use up all the representing array—we take indices modulo the length of the array

For **deques** we can proceed analogously combining the operations of stacks and queues; we omit further details

FIFO Queue Operations

The (most important) operations on queues are the following:

- ***isEmpty*** returns **true**, if the queue contains no element, and **false** otherwise—the complexity is clearly in $\Theta(1)$
- ***popfront*** removes and returns the front element of a non-empty queue; it is undefined for empty queues—the (amortised) complexity (on circular arrays) is clearly in $\Theta(1)$
- ***front*** returns the front element of a non-empty queue without changing the queue; it is undefined for empty queues—the complexity (on circular arrays) is clearly in $\Theta(1)$
- ***pushback*** adds a new element to the end of the queue—the (amortised) complexity is clearly in $\Theta(1)$
- ***back*** returns the back element of a non-empty queue without changing the queue; it is undefined for empty queues—the complexity is clearly in $\Theta(1)$