CS 225 – Data Structures

ZJUI – Spring 2022

Lecture 3: Algorithms on Sequences

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

3 Algorithms on Lists

3.1 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

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

Mergesort / cont.

merged_list $\leftarrow$ *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 $\leftarrow$ *merge*(restlist$_1$, inlist$_2$) **ENDIF**
IF $x_1 > x_2 \wedge$ merged_restlist = *undef*
THEN merged_restlist $\leftarrow$ *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
[8], [13], [5], [1], [2], [4], [3], [7], [6]	1+1+1+1+1+1+1+1+1 (sorted)
[8, 13], [1, 5], [2, 4], [3], [6, 7]	2+2+2+1+2 (sorted)
[1, 5, 8, 13], [2, 4], [3, 6, 7]	4+2+3 (sorted)
[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 $\lceil n \log_2 n \rceil$ 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 $x_1, \ldots, x_n$

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 $x_{\sigma(1)}, \ldots, x_{\sigma(n)}$

If $v_0, \ldots, v_d$ 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 $\varphi_k = \begin{cases} x_{k_i} \leq x_{k_j} & \text{if } v_{k+1} \text{ is the left successor of } v_k \\ x_{k_j} < x_{k_i} & \text{if } v_{k+1} \text{ is the right successor of } v_k \end{cases}$
- $\{\varphi_0, \ldots, \varphi_{d-1}\}$ imply $x_{\sigma(1)} \leq \cdots \leq x_{\sigma(n)}$

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.

$$
\begin{array}{l}
\text{sorted\_list} \leftarrow \mathit{qsort}(\text{unsorted\_list}) = \\
\quad \textbf{PAR} \;\; \textbf{IF } \text{unsorted\_list} = [] \textbf{ THEN } \text{sorted\_list} := []) \textbf{ ENDIF} \\
\qquad\quad \textbf{IF } \text{unsorted\_list} \neq [] \wedge \text{sorted\_list}_1 = \mathit{undef} \wedge \text{sorted\_list}_2 = \mathit{undef} \\
\qquad\quad \textbf{THEN} \;\; \textbf{CHOOSE } x \in \text{unsorted\_list } \textbf{DO} \\
\qquad\qquad\qquad\quad \text{pivot} := x \\
\qquad\qquad\qquad\quad \textbf{LET } \text{unsorted\_list}_1 = \text{sublist}[y \mid y < x](\text{unsorted\_list}) \textbf{ IN} \\
\qquad\qquad\qquad\quad \textbf{LET } \text{unsorted\_list}_2 = \text{sublist}[y \mid y > x](\text{unsorted\_list}) \textbf{ IN} \\
\qquad\qquad\qquad\quad \textbf{PAR} \;\; \text{sorted\_list}_1 \leftarrow \mathit{qsort}(\text{unsorted\_list}_1) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{sorted\_list}_2 \leftarrow \mathit{qsort}(\text{unsorted\_list}_2))) \\
\qquad\qquad\qquad\quad \textbf{ENDPAR} \\
\qquad\quad \textbf{ENDIF} \\
\qquad\quad \textbf{IF } \text{unsorted\_list} \neq [] \wedge \text{sorted\_list}_1 \neq \mathit{undef} \wedge \text{sorted\_list}_2 \neq \mathit{undef} \\
\qquad\quad \textbf{THEN} \;\; \text{sorted\_list} := \text{concat}(\text{sorted\_list}_1, \\
\qquad\qquad\qquad\qquad\quad \text{concat}([\text{pivot}], \text{sorted\_list}_2)) \\
\qquad\quad \textbf{ENDIF} \\
\quad \textbf{ENDPAR}
\end{array}
$$

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 $C(n) \leq \frac{n(n-1)}{2}$

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 $x_1, \ldots, x_n$ 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 $M = \{x_i, \ldots, x_j\}$, 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 instance for $T = \mathbb{N}$, because then we can exploit arithmetic operators on T

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

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 $key(List(i)) \leq key(List(j))$

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 $key(t) \in \{0, \ldots, K-1\}$ 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 $list_0, \ldots, list_{K-1}$ 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 $0, \ldots, K-1$), and that we need to represent key values in the range $0, \ldots, K^d - 1$

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

3.2 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

3.3 Divide and Conquer

A **divide-and-conquer algorithm** is an algorithm that

- decomposes the given problem instance into smaller subinstances of the same problem,
- solves each of these subproblems independently,
- combines the solutions of the subproblems into a solution of the given problem instance

In order to solve a subproblem, either the divide-and-conquer algorithm is applied recursively or a (simpler) base algorithm is used

The decision whether to use recursion or a different base algorithm usually depends on the size of the problem, for which a threshold value is used

For the divide-and-conquer technique we usually assume that the decomposition leads to at least two subproblems—the case where only a single subinstance is to be solved is called **simplification**

Selection and the Median

If we have a list of elements in some ordered set T, then it is easy to determine the minimum or the average of the list elements; this can be done in linear time by scanning the list

If the list ℓ has n elements, then the **median** is the element $m \in T$ such that

$$|\{i \in \{1, \ldots, n\} \mid \ell(i) < m\}| < n/2 \quad \text{and} \quad |\{i \in \{1, \ldots, n\} \mid \ell(i) \leq m\}| \geq n/2$$

So roughly speaking, the median is the element m of the list for which less than half of the list elements are smaller and at most half of the elements are not smaller than m

A naive algorithm could just sort the list and then retrieve the element in position $\lceil n/2 \rceil$

We will now explore a better solution using divide-and-conquer, where the ideas are borrowed from quicksort

The Selection Problem

Let us consider the slightly more general **selection problem**: for $1 \leq k \leq n$ find the k'th smallest element in a list ℓ with n elements

For $k = 1$ this means to find the minimum, for $k = \lceil n/2 \rceil$ the problem is to find the median

Analogously to quicksort we can proceed as follows:

- choose an arbitrary element p from the list, the **pivot** element
- split the list into the sublist U containing the u elements $< p$, the sublist V containing the $n - v$ elements $> p$ and the rest
- if $k \leq u$ holds, we have to search for the k'th smallest element in U
- if $k > v$ holds, we have to search for the $(k - v)$'th smallest element in V
- otherwise, the sought k'th smallest element is the pivot p

The Selection Algorithm

result $\leftarrow$ SELECT$(k, list)$

CHOOSE pivot **WITH** member(pivot, $list$)

DO

LET $u = |\{i \mid 1 \leq i \leq |\ell| \wedge list(i) < \text{pivot}\}|$,

$v = |\{i \mid 1 \leq i \leq |\ell| \wedge list(i) \leq \text{pivot}\}|$

IN PAR

IF $k \leq u$ **THEN**

LET $U = \mathbf{I}\ell.\forall x.(member(x, \ell) \leftrightarrow member(x, list) \wedge x < \text{pivot})$

IN result $\leftarrow$ SELECT(k, U)

ENDIF

IF $u < k \leq v$ **THEN** result := pivot **ENDIF**

IF $k > v$ **THEN**

LET $V = \mathbf{I}\ell.\forall x.(member(x, \ell) \leftrightarrow member(x, list) \wedge x > \text{pivot})$

IN result $\leftarrow$ SELECT$(k - v, V)$

ENDIF

ENDPAR

ENDDO

Worst Case Complexity

The algorithm above only considers the core of the solution to the selection problem

In addition we need a base algorithm, which is used, if the size of the input list is smaller than some threshold

- We could use sorting and linear search to find the $\lceil n/2 \rceil$'th smallest element, if $n \leq thr$ holds
- For $n = 1$ we may simply return the only element of the list

The worst case complexity occurs, when all elements of the list are pairwise different and either the minimum or the maximum is always chosen as the pivot element

In the former case we get $U = \emptyset$ and $|V| = n - 1$; in the latter case we get $V = \emptyset$ and $|U| = n - 1$

In both cases the rule is iterated $n - 1$ times, each time scanning the whole input list to decompose it into U, V and the rest

Hence the worst case complexity is in $O(n^2)$

Average Case Complexity

On average the time complexity of $\textsc{Select}(k, list)$ is in $O(|list|)$.

Proof. Let $X(n,k)$ be the size of the sublist in the first recursive call of the algorithm (i.e., it is $|U|$ or $|V|$)—with the proviso that $X(n,k) = 0$, if there is no such call

In particular, we have $X(n,k) = 0$ for $k \geq n$, in which case we either search the maximum in time $O(n)$ or return an error message, as a list with n elements does not contain a k'th smallest element, if $k > n$ holds

As X is a random variable, we need to consider its expectation value $E(n,k) = E(X(n,k))$

Assume that the list elements are $x_1 < \cdots < x_n$

Then we have $$E(n,k \mid \text{pivot} = x_i) = \begin{cases} i-1 & \text{if } k < i-1 \\ 0 & \text{if } k = i \\ n-i & \text{if } k > i \end{cases}$$

Proof (cont.)

This gives

$$\begin{aligned} E(n,k) &= \frac{1}{n}\sum_{i=1}^{n} E(n,k \mid \text{pivot} = x_i) = \frac{1}{n}\left(\sum_{i=k+2}^{n}(i-1) + \sum_{i=1}^{k-1}(n-i)\right) \\ &= \frac{1}{n}\left(\frac{(n-1)n}{2} - \frac{k(k+1)}{2} + n(k-1) - \frac{k(k-1)}{2}\right) \\ &= \frac{n-1}{2} - \frac{k^2}{n} + (k-1) < \frac{n}{2} + \frac{k(n-k)}{n} \leq \frac{3}{4}n \end{aligned}$$

as $k(n-k)$ is maximal for $k = n-k$, i.e. for $k = n/2$

Each iteration of the rule $\textsc{Select}(k, list)$ has to scan the input list, compute the values u and v, build the sets U and V, and either recursively call either $\textsc{Select}(k, U)$ or $\textsc{Select}(n-k, V)$ or execute the base algorithm, hence its complexity is in $\Theta(|list|)$

If we have m iterations of the rule, then the expected complexity is bounded by

$$c \cdot \sum_{i=0}^{m} \left(\frac{3}{4}\right)^i \cdot n < \frac{1}{1-\frac{3}{4}} \cdot c \cdot n = 4cn \in O(n) \ ,$$

i.e., the average case complexity is linear as claimed

Matrix Multiplication

The multiplication of two matrices with elements in $\mathbb{R}$ or $\mathbb{C}$ is a common operation in numerical mathematics and its applications in science and engineering

Let us concentrate here on quadratic matrices, i.e $(n \times n)$ matrices:

$$\left(a_{i,j}\right)_{1 \le i,j \le n} \cdot \left(b_{i,j}\right)_{1 \le i,j \le n} = \left(c_{i,j}\right)_{1 \le i,j \le n}$$

with $c_{i,k} = \sum_{j=1}^{n} a_{i,j} b_{j,k}$

So we have to compute n^2 elements of the product matrix $C = A \cdot B$; for each such element we need $(n-1)$ additions and n multiplications of elements in K ($K = \mathbb{R}$ or $K = \mathbb{C}$)

So, with just the application of the definition we get $n^2(2n-1)$ elementary operations in K, which gives us a time complexity in $\Theta(n^3)$

Here we count addition and multiplication in K as "elementary" with constant time complexity in $\Theta(1)$

The Base Case: $n = 2$

In order to improve this complexity we look into the **Strassen algorithm** (1960)

Let us first consider the case $n = 2$:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_2 + c_3 & (c_1 + c_2) + (c_5 + c_6) \\ ((c_1 + c_2) + c_4) - c_7 & ((c_1 + c_2) + c_4) + c_5 \end{pmatrix}$$

where the c_i are computed as follows:

$$\begin{aligned} c_1 &= c'_1 c'_2 & c'_1 &= c'_5 - a_{11} \\ c_2 &= a_{11} b_{11} & c'_2 &= c'_4 + b_{11} \\ c_3 &= a_{12} b_{21} & & \\ c_4 &= (a_{11} - a_{21}) c'_4 & c'_4 &= b_{22} - b_{12} \\ c_5 &= c'_5 (b_{12} - b_{11}) & c'_5 &= a_{21} + a_{22} \\ c_6 &= (a_{12} - c'_1) b_{22} & & \\ c_7 &= a_{22} (c'_2 - b_{21}) & & \end{aligned}$$

Here we count 15 additions and 7 multiplications—compared to the 12 elementary operations when using just the definition this is no improvement at all

Divide and Conquer

However, the same calculation also applies, when the $a_{i,j}$ and $b_{i,j}$ are $(m \times m)$ matrices (so we multiply two $(2m \times 2m)$ matrices

In this case we have 15 matrix additions and 7 matrix multiplications, and the addition of two $(m \times m)$ matrices requires only m^2 scalar operations (so it is much more efficient than matrix multiplication)

This gives rise to a divide-and-conquer strategy: if n is a power of 2, say $n = 2^m$, use the equations above to reduce matrix multiplication to 7 matrix multiplications of size 2^{m-1} plus 15 matrix additions

If $MP(n)$ is the number of elementary scalar operations (addition, multiplication) needed to compute the product $A \cdot B$ of $(n \times n)$ matrices, then we have

$$MP(2^n) \;=\; 7MP(2^{n-1}) + 15AP(2^{n-1}) \;=\; 7MP(2^{n-1}) + 15 \cdot 2^{2n-2} \;,$$

where $AP(n)$ is the number of scalar operations needed to compute the product $A + B$ of $(n \times n)$ matrices

If $n \leq thr$ for some threshold value thr, switch to the basic algorithm, i.e. apply directly the definition to compute the matrix product

Worst Case Complexity

If we replace 2^n by n, then the above equation for MP gives rise to the recurrence equation $mp_n = 7mp_{n-1} + 15/4n^2$

Using the common method to solve linear recurrence equations we take the characteristic polynomial $(x-7)(x-1)^3$ with the roots 7 and 1 (multiplicity 3)

The general form of the solution is $7^n c_1 + \frac{1}{2}(n^2 - n)c_2 + c_3 n + c_4$

Replacing 2^n again by n we get the solution

$$n^{\log_2 7} c_1 + c_2'(\log n)^2 + c_3' \log n + c_4 \in \Theta(n^{\log_2 7})$$

with $n^{\log_2 7} \approx 2.807355 \approx 2.81$

The General Case

The remaining question concerns the generalisation to arbitrary $(n \times n)$ matrices, when n is not a power of 2

Variant 1. If n is odd, write an $(n \times n)$ matrix as $\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$, where $A_{11}, A_{12}, A_{21}, A_{22}$ are $(\lfloor n/2 \rfloor \times \lfloor n/2 \rfloor)$ matrices, A_{13}, A_{23} are $(\lfloor n/2 \rfloor \times 1)$ matrices, A_{31}, A_{32} are $(1 \times \lfloor n/2 \rfloor)$ matrices, and A_{33} is a scalar

Then write $MP(n) = 7MP(\lfloor n/2 \rfloor) + 15AP(\lfloor n/2 \rfloor) + O(n)$, which leads to a slightly modified recurrence equation leading again to complexity in $\Theta(n^{2.81})$

Variant 2. Extend the $(n \times n)$ matrices to $(2^m \times 2^m)$ matrices with minimal m (just add columns and rows with 0), and apply the algorithm—many computations can be ignored, as they result in 0

This gives us complexity in $\Theta((2n)^{2.81}) = \Theta(n^{2.81})$

List Rotation

Let ℓ be a list of length n, and let $\ell_1 = [\ell(i), \ldots, \ell(i+m-1)]$ and $\ell_2 = [\ell(j), \ldots, \ell(j+m-1)$ be two sublists of length m

First consider the operation $exchange(i, j, m)$ to swap these two sublists—so we require $i + m \leq j \leq n - m + 1$ to avoid ambiguity

If we take doubly linked lists, it will take time in $\Theta(1)$, as only a few pointers have to be rearranged

As the length of the list is invariant under this operation, we can further assume without loss of generality that we deal with an array instead of a list

Then it is clear that the operation can be done in $\Theta(m)$ time, as we have to swap $\ell(i+k)$ with $\ell(j+k)$ for all $0 \leq k \leq m-1$

The Rotate Operation

Now consider the related operation $rotate(m)$, which rotates the list by m position, i.e. the result is the list ℓ' with $\ell'(i) = \ell((i + m - 1) \bmod n)$

We want to obtain an algorithm for $rotate(m)$ that does not need an additional array

For this we successively reduce the rotation problem to smaller and smaller sublists each time exploiting the *exchange* operation

We initialize $k = 1$, $i = m$ and $j = n - m$, so that the sublist of interest in each step will be $[\ell(k), \ldots, \ell(k + i + j - 1)]$

In the following algorithm the interval $[\min(i, j), \max(i, j)]$ shrinks in each iteration step (except the final one), so the algorithm will terminate

We use another location halt, which is initialised to false

```
IF halt = false
    par   if i > j then par
                            exchange(k, k + i, j)
                            i := i − j
                            k := k + j
                        endpar
          endif
          if i < j then par
                            exchange(k, k + j, i)
                            j := j − i
                        endpar
          endif
          if i = j then par
                            exchange(k, k + i, i)
                            halt := true
                        endpar
          endif
    endpar
endif
```

Correctness

Consider the sublist $[\ell(k), \ldots, \ell(k+i+j-1)]$, which initially is the whole given list ℓ

Every iteration step only changes this sublist, and after each iteration step we have

- the list elements outside this sublist are already in the correct place as requested for the rotation operation
- the operation $rotate(i)$ needs to be executed on this sublist

This follows by induction: the induction base is trivial, and for the induction step observe that in all three cases ($i > j$, $i < j$ and $i = j$) the initial i or j elements (respectively) are swapped with the last i or j elements

Furthermore, the *exchange* in the last step swaps two equally-sized sections of the array, so it is de fact a *rotate* operation

Complexity

Let $T(i,j)$ denote the number of elementary swaps to execute $rotate(i)$ on a list of length $n = i + j$

In our algorithm we have $T(i,j) = \begin{cases} i & \text{if } i = j \\ j + T(i-j, j) & \text{if } i > j \\ i + T(i, j-i) & \text{if } i < j \end{cases}$

Then we get $T(i,j) = i + j - \gcd(i,j)$, which we prove by induction over the number of steps in the Euclidean algorithm

For the induction base we have $i = j$ and $\gcd(i,j) = j$

For the induction step, if $i > j$, we have $T(i,j) = j + T(i-j,j) \stackrel{i.h.}{=} j + i + j - \gcd(i-j,j) = i + j - \gcd(i,j)$

Analogously, for $i < j$ we get $T(i,j) = i + T(i,j-i) \stackrel{i.h.}{=} j + i + j - \gcd(i,j-i) = i + j - \gcd(i,j)$

Hence, for $rotate(m)$ on a list with n elements the number of elementary swaps is $n - \gcd(n,m)$; **worst case** $\Theta(n)$ and **best case** $\Theta(n-m)$