CS 225 – Data Structures

ZJUI – Spring 2021

Lecture 3: Divide and Conquer

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

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

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**

3.1 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

3.2 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 $\Theta((2n)^{2.81}) = \Theta(n^{2.81})$

3.3 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** $\Theta(n-m)$