Selected Topics in Algorithms

Divide and Conquer

Marcin Sydow

"Divide and Conquer" and Searching

`search(S, len, key)`
(input sequence is sorted)

The Binary Search Algorithm (the "Divide and Conquer" approach)

1. while the length of sequence is positive:
2. check the middle element of the current sequence
3. if it is equal to key - return the result
4. if it is higher than key - restrict searching to the "left" sub-sequence (from the current position)
5. if it is less than key - restrict searching to the "right" sub-sequence (from the current position)
6. back to the point 1
7. there is no key in the sequence (if you are here)

Binary Search Algorithm


```
search(S, len, key){

  l = 0
  r = len - 1

  while(l <= r){
      m = (l + r)/2
      if(S[m] == key) return m
      else
        if(S[m] > key) r = m - 1
        else l = m + 1
  }

  return -1
}
```

Notice that the operation of **random access** (direct access) to the m-th element `S[m]` of the sequence demands that the sequence is kept in RAM (to make the operation efficient)

Recursion

e.g.: $n! = (n-1)!n$

- Mathematics: recurrent formula or definition
- Programming: function that calls itself
- Algorithms: reduction of an instance of a problem to a smaller instance of the same problem ("divide and conquer")

Warning: should be well founded on the trivial case:

e.g.: $0! = 1$

Example

step:

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)

base:

Fibonacci(0) = Fibonacci(1) = 1

1,1,2,3,5,8,13,21,34,...

Recursion as an Algorithmic Tool

A powerful method for algorithm design

It has positive and negative aspects, though:

- (positive) very compact representation of an algorithm
- (negative) recursion implicitly costs additional memory for keeping the recursion stack

Example

What happens on your machine when you call the following function for n=100000?

```
triangleNumber(n){
 if (n > 0) return triangleNumber(n-1) + n
 else return 0
}
```

Iterative version of the above algorithm would not cause any problems on any reasonable machine.

In final implementation, recursion should be avoided or translated to iterations whenever possible (not always possible), due to the additional memory cost for keeping the recursion stack (that could be fatal...)

Hanoi Towers

A riddle:

Three vertical sticks A, B and C. On stick A, stack of n rings, each of different size, always smaller one lies on a bigger one. Move all rings one by one from A to C, respecting the following rule "bigger ring cannot lie on a smaller one" (it is possible to use the helper stick B)

Hanoi Towers - number of moves

How many moves are needed for moving n rings?
(hanoi(n) = ?)

This task can be easily solved with recurrent approach.

Hanoi Towers - number of moves

How many moves are needed for moving n rings?
(hanoi(n) = ?)

This task can be easily solved with recurrent approach.

If we have 1 ring, we need only 1 move (A -> C). For more rings, if we know how to move n-1 top rings to B, then we need to move the largest ring to C, and finally all rings from B to C.

Hanoi Towers - number of moves

How many moves are needed for moving n rings?
(hanoi(n) = ?)

This task can be easily solved with recurrent approach.

If we have 1 ring, we need only 1 move (A -> C). For more rings, if we know how to move n-1 top rings to B, then we need to move the largest ring to C, and finally all rings from B to C.

Thus, we obtain the following recurrent equations:
base:
hanoi(1) = 1

step:
hanoi(n) = hanoi(n-1) + 1 + hanoi(n-1) = 2*hanoi(n-1) + 1

Sorting

Input: S - sequence of elements that can be ordered (according to some binary total-order relation $R \subseteq S \times S$); len - the length of sequence (natural number)

Output: S' - non-decreasingly sorted sequence consisting of elements of multi-set of the input sequence S (e.g. $\forall_{0<i<len}(S[i-1], S[i]) \in R$)

In this course, for simplicity, we assume sorting natural numbers, but all the discussed algorithms which use comparisons can be easily adapted to sort any other ordered universe.

The Importance of Sorting

Sorting is one of the most important and basic operations in any real-life data processing in computer science. For this reason it was very intensively studied since the half of the 20th century, and currently is regarded as a well studied problem in computer science.

Examples of very important applications of sorting:

- acceleration of searching
- acceleration of operations on relations "by key", etc. (e.g. in databases)
- data visualisation
- computing many important statistical characteristics

And many others.

Selection Sort

The idea is simple. Identify the minimum (*len* times) excluding it from the further processing and putting on the next position in the output sequence:

```
selectionSort(S, len){
  i = 0
  while(i < len){
    mini = indexOfMin(S, i, len)
    swap(S, i, mini)
    i++
  }
}
```

where:

`indexOfMin(S, i, len)` - return index of minimum among the elements $S[j]$, where $i \leq j < len$

`swap(S, i, mini)` - swap the positions of S[i] and S[mini]

What is the invariant of the above loop?

Insertion Sort


```
insertionSort(arr, len){

  for(next = 1; next < len; next++){

    curr = next;
    temp = arr[next];

    while((curr > 0) && (temp < arr[curr - 1])){

      arr[curr] = arr[curr - 1];
      curr--;
    }

    arr[curr] = temp;
  }
}
```

What is the invariant of the external loop?

Insertion Sort - Analysis

(dominating operation and data size n is the same for all the algorithms discussed in this lecture)

What is the pessimistic case?

Insertion Sort - Analysis

(dominating operation and data size n is the same for all the algorithms discussed in this lecture)

What is the pessimistic case?

When the data is invertedly sorted. Then the complexity is:

$$W(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 + \Theta(n) = \Theta(n^2)$$

This algorithm is much more "intelligent" than the previous, because it adapts the amount of work to the degree of sortedness of the input data - the more sorted input the less number of comparisons (and swaps). In particular, for **already sorted** data it needs only n-1 comparisons (is linear in this case - very fast!).

Average Time Complexity Analysis

Let's assume a simple model of input data - each permutation of input elements is equally likely. Then, for i-th iteration of the external loop the algorithm will need (on average):

$$\frac{1}{i}\sum_{j=1}^{i} j = \frac{1}{i}\frac{(i+1)i}{2} = \frac{i+1}{2}$$

comparisons. Thus, we obtain:

$$A(n) = \sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2}\sum_{k=2}^{n} k = \frac{1}{4}n^2 + \Theta(n) = \Theta(n^2)$$

Divide and conquer sorting (1) - Merge Sort

Let's apply the "divide and conquer" approach to the sorting problem.

1. divide the sequence into 2 halves
2. **sort** each half separately
3. merge the sorted halves

This approach is successful because sorted subsequences can be merged very quickly (i.e. with merely linear complexity)

Moreover, let's observe that sorting in point 2 can be recursively done with the same method (until the "halves" have zero lengths)

Thus, we have a working recursive sorting scheme (by merging).

Merge Sort - Scheme


```
mergeSort(S, len){
  if(len <= 1) return S[0:len]
  m = len/2
  return merge(mergeSort(S[0:m], m), m
               mergeSort(S[m:len], len-m), len-m)
}
```

where:

- denotation S[a:b] means the subsequence of elements S[i] such that $a \leq i < b$
- the function `merge(S1, len1, S2, len2)` merges 2 (sorted) sequences S1 and S2 (of lengths len1 and len2) and **returns** the merged (and sorted) sequence.

Merge Function

input: a1, a2 - sorted sequences of numbers (of lengths len1, len2)
output: return merged (and sorted) sequences a1 and a2

```
merge(a1, len1, a2, len2){

  i = j = k = 0;
  result[len1 + len2] // memory allocation

  while((i < len1) && (j < len2))
    if(a1[i] < a2[j]) result[k++] = a1[i++];
    else result[k++] = a2[j++];

  while(i < len1) result[k++] = a1[i++];

  while(j < len2) result[k++] = a2[j++];

  return result;
}
```

Quick Sort - idea

Quick sort is based on the “divide and conquer” approach.

The idea is as follows (recursive version):

1. For the sequence of length 1 nothing has to be done (stop the recursion)
2. longer sequence is reorganised so that some element M (called “pivot”) of the sequence is put on “final” position so that there is no larger element “to the left” of M and no smaller element “to the right” of M.
3. subsequently steps 1 and 2 are applied to the “left” and “right” subsequences (recursively)

The idea of quick sort comes from C.A.R.Hoare.

Partition procedure - reminder

`partition(S, l, r)`
For a given sequence S (bound by two indexes l and r) the `partition` procedure **selects** some element M (called "pivot") and efficiently reorganises the sequence so that M is put on such a "final" position so that there is no larger element "to the left" of M and no smaller element "to the right" of M.
The partition procedure **returns** the final index of element M.

For the following assumptions:

- **Dominating operation:** comparing 2 elements
- **Data size**: the length of the array $n = (r - l + 1)$

The `partition` procedure can be implemented so that it's time complexity is $W(n) = A(n) = \Theta(n)$ and space complexity is $S(n) = O(1)$

Partition - possible implementation

input: a - array of integers; l,r - leftmost and rightmost indexes, respectively;
output: the final index of the "pivot" element M; the side effect: array is reorganised (no larger on left, no smaller on right)

```
partition(a, l, r){

  i = l + 1;
  j = r;
  m = a[l];
  temp;

  do{
    while((i < r) && (a[i] <= m)) i++;
    while((j > i) && (a[j] >= m)) j--;
    if(i < j) {temp = a[i]; a[i] = a[j]; a[j] = temp;}
  }while(i < j);
  // when (i==r):
  if(a[i] > m) {a[l] = a[i - 1]; a[i - 1] = m; return i - 1;}
  else {a[l] = a[i]; a[i] = m; return i;}
}
```

QuickSort - pseudo-code

Having defined `partition` it is now easy to write a recursive QuickSort algorithm described before:

input: a - array of integers; l,r - leftmost and rightmost indexes of the array

(the procedure does not return anything)

```
quicksort(a, l, r){

    if(l >= r) return;
    k = partition(a, l, r);
    quicksort(a, l, k - 1);
    quicksort(a, k + 1, r);
}
```

Solving Recurrent Equations

2 general methods:

1. expanding to sum
2. generating functions

illustration of the method 1:

$hanoi(n) = 2 * hanoi(n-1) + 1 =$
$2 * (2 * hanoi(n-2) + 1) + 1 = ... = \sum_{i}^{n-1} 2^i = 2^n - 1$

(method 2 is outside of the scope of this course)

A general method for solving 2nd order linear recurrent equations

Assume the following recurrent equation:

$s_n = as_{n-1} + bs_{n-2}$

Then, solve the following *characteristic equation*:
$x^2 - ax - b = 0$.

1. single solution r: $s_n = c_1 r^n + c_2 n r^n$
2. two solutions r_1, r_2: $s_n = c_1 r_1^n + c_2 r_2^n$

for some constants c_1, c_2
(that can be found by substituting $n = 0$ and $n = 1$)

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

From the last theorem it can be shown that:

$$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{n+1} - (\frac{1-\sqrt{5}}{2})^{n+1})$$

(the "Euler-Binet" formula)

(BTW: it is incredible, but this is always a natural number!)

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

From the last theorem it can be shown that:

$$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{n+1} - (\frac{1-\sqrt{5}}{2})^{n+1})$$

(the “Euler-Binet” formula)

(BTW: it is incredible, but this is always a natural number!)

Lets guess what a number is Fibonacci(50)...

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

From the last theorem it can be shown that:

$$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{n+1} - (\frac{1-\sqrt{5}}{2})^{n+1})$$

(the "Euler-Binet" formula)

(BTW: it is incredible, but this is always a natural number!)

Lets guess what a number is Fibonacci(50)...
it is precisely 12 586 269 025 (over 12 billion!)

Other Important Special Cases

Some types of recurrent equations are quite frequently encountered in algorithmics.

I.e. time complexity function of some important algorithms is in the form of a recurrent equation of such type

We show 3 of them with simple solutions (on rank of complexity)

Case 1

t(1) = 0
t(n) = t(n/2) + c; n>0, $c \in N$ is a constant
($n/2$ means $\lfloor(n/2)\rfloor$ or $\lceil(n/2)\rceil$)

example of algorithm?

Case 1

t(1) = 0
t(n) = t(n/2) + c; n>0, $c \in N$ is a constant
($n/2$ means $\lfloor(n/2)\rfloor$ or $\lceil(n/2)\rceil$)

example of algorithm?

proof: (substitute $n = 2^k$)

$t(2^k) = t(2^{k-1}) + c = t(2^{k-2}) + c + c = t(2^0) + kc = kc = clog(n)$

solution: $t(n) = c(log(n)) = \Theta(log(n))$ (logarithmic)

example of algorithm:
binSearch (a version that assumes that the sequence contains the key, since $t(1) = 0$)

Case 2

$t(1) = 0$

$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + c;\ n>0,\ c \in N$ is a constant

example of algorithm?

Case 2

t(1) = 0
t(n) = t($\lfloor(n/2)\rfloor$) + t($\lceil(n/2)\rceil$) + c; n>0, $c \in N$ is a constant

example of algorithm?

proof: (substitute $n = 2^k$)

$t(2^k) = 2t(2^{k-1}) + c = 2(2t(2^{k-2}) + c) + c =$
$2^2(t(2^{k-2})) + 2^1c + 2^0c =$
$2^k t(2^0) + c(2^{k-1} + 2^{k-2} + \ldots + 2^0) = 0 + c(2^k - 1) = c(n-1)$

solution: $t(n) = c(n-1) = \Theta(n)$ (linear)

example: maximum in sequence

Case 3

t(1) = 0

t(n) = t($\lfloor(n/2)\rfloor$) + t($\lceil(n/2)\rceil$) + cn; n>0, $c \in N$ is a constant

example of algorithm?

Case 3

t(1) = 0

t(n) = t($\lfloor(n/2)\rfloor$) + t($\lceil(n/2)\rceil$) + cn; n>0, $c \in N$ is a constant

example of algorithm?

proof: (substitute $n = 2^k$)

$t(2^k) = 2t(2^{k-1}) + c2^k = 2(2t(2^{k-2}) + c2^{k-1}) + c2^k = 2^2 t(2^{k-2}) + c2^k + c2^k = 2^k t(2^0) + kc2^k = 0 + cnlog(n)$

solution: $cn(log(n)) = \Theta(nlog(n))$ (linear-logarithmic)

example of algorithm: mergeSort

Completing the Proofs

We solved the equations only for exact powers of 2, i.e. $n = 2^k$. The asymptotic bounds, however, will hold in general, due to the following lemma:

If non-decreasing functions: $t(n) : N \to N$ and $f(x) : R \to R$ satisfy:

- $t(2^k) = \Theta(f(2^k))$, for $k \in N$
- $\exists_{x_0>0}\exists_{c>0}\forall_{x \geq x_0} f(2x) \leq cf(x)$

Then $t(n) = \Theta(f(n))$.

What functions satisfy the second condition? $(x, logx, xlogx, x^2, 2^x)$?

Simple proofs presented on the last few slides are based on: Banachowski, Diks, Rytter "Introduction to Algorithms", Polish 3rd Edition, WNT, 2001, pp.20-21 and p.43; (BDR)

Example - the Average Quicksort's Complexity

Lets solve the following recurrent equation:
$A(0) = A(1) = 0$
$A(n) = (n+1) + \frac{1}{n}(\sum_{s=1}^{n}(A(s-1) + A(n-s))); n > 1$

(The equation represents the average time complexity of some version of quickSort, that can be found e.g. in BDR, with assumption that input data is uniformly distributed among all permutations of n elements)

$$A(n) = \frac{2}{n}\sum_{s=1}^{n} A(s-1) + (n+1)$$

Transform the above to the two following equations:
$nA(n) = 2\sum_{s=1}^{n} A(s-1) + n(n+1)$
$(n-1)A(n-1) = 2\sum_{s=1}^{n-1} A(s-1) + (n-1)n$

Average QuickSort's Complexity, cont.

Lets subtract the 2nd equation from the first:

$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2n$

$nA(n) = (n+1)A(n-1) + 2n$

$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1}$

Now, lets expand the last equation:

$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1} = \frac{a(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} =$

$= \frac{A(1)}{2} + 2/3 + 2/4 + ... + \frac{2}{n+1} =$

$2(1 + 1/2 + 1/3 + ... + 1/(n+1) - 3/2)$

Thus,

$A(n) = 2(n+1)(1 + 1/2 + 1/3 + ... + 1/(n+1) - 3/2)$

Harmonic Number (cont. of the proof)

$A(n) = 2(n+1)(1 + 1/2 + 1/3 + ... + 1/(n+1) - 3/2)$

The sum $1 + 1/2 + 1/3 + ... + 1/n$ is called the n-th *harmonic number*, denoted as H_n

It can be proved that asymptotically the following holds:

$H_n = ln(n) + \gamma + O(n^{-1})$, where $\gamma \approx 0,5772156...$ is called the *Euler's constant*.

Thus, finally we obtain:

$A(n) = (\frac{2}{log(e)})(n+1)log(n) + O(n) = \frac{2}{log(e)}nlog(n) + O(n) = \Theta(nlog(n))$ (the factor $2/log(e) \approx 1.44$)

This ends the proof of $\Theta(nlog(n))$ average time complexity of quickSort

Master Theorem - Introduction

(Pol.: “twierdzenie o rekurencji uniwersalnej”)

A universal method for solving recurrent equations of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1, b > 1$: *constants*, $f(n)$ is asymptotically positive

It can represent time complexity of a recurrent algorithm that divides a problem to a sub-problems, each of size n/b and then merges the sub-solutions with the additional complexity described by $f(n)$

E.g. for mergeSort $a = 2, b = 2, f(n) = \Theta(n)$

Master Theorem (Pol.: "Twierdzenie o rekurencji uniwersalnej")

Assume, $T(n): N \rightarrow R$ is defined as follows:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1, b > 1$: *constants*, n/b denotes $\lfloor (n/b) \rfloor$ or $\lceil (n/b) \rceil$ and $f(n): R \rightarrow R$ is asymptotically positive

Then $T(n)$ can be asymptotically bounded as follows:

1. if $f(n) = O(n^{log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$
2. if $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a} lgn)$
3. if $f(n) = \Omega(n^{log_b a + \epsilon})$ for some $\epsilon > 0$, and if asymptotically $af(n/b) \leq cf(n)$ for some $c < 1$ ("regularity" condition), then $T(n) = \Theta(f(n))$

(Proof in CLR 4.4)

Interpretation and "Gaps"in the Master Theorem

Lets interpret the Master Theorem. To put it simply, it compares $f(n)$ with $n^{log_b a}$ and states that the function of the higer rank of complexity determines the solution:

1. if $f(n)$ is of *polynomially* lower rank than $n^{log_b a}$, the latter dominates
2. if $f(n)$ and $n^{log_b a}$ are of the same rank, the lgn coefficient occurs
3. if $f(n)$ is of *polynomially* higher rank than $n^{log_b a}$ and satisfies the “regularity” condition, the former function represents the rank of complexity

Some cases are not covered by the Master Theorem, i.e. for functions $f(n)$ that fall into “gaps” between conditions 1-2 or 2-3 or that do not satisfy the “regulartity” condition. In such cases the theorem cannot be applied.

Thank you for your attention!