Chair for Algorithms
and Data Structures
Prof. Dr. Hannah Bast
Claudius Korzen

Information Retrieval
WS 2018/2019

http://ad-wiki.informatik.uni-freiburg.de/teaching

UNI FREIBURG

Exam

Tuesday, February 12, 2019, 14:00 - 16:30, Lecture Hall 101-026

General instructions:

There are six tasks, of which you can select *five tasks of your choice*. Each task is worth 20 points. If you do all six tasks, we will only count the best five, that is, you can reach a maximum number of 100 points.

You need 50 points to pass the exam. You have 150 minutes of time overall. If you do five tasks, this is 30 minutes per task on average.

You are allowed to use any amount of paper, books, etc. You are not allowed to use any computing devices or mobile phones, in particular nothing with which you can communicate with others or connect to the Internet or parallel universes.

You may write down your solutions in either English or German.

Please write your solutions on this hand-out, below the description of the tasks! You can also use the back side of the pages. Please write your name and Matrikelnummer on the top of this cover sheet in the framed box. If you need additional pages, please write your name and Matrikelnummer on each of them, too.

Important:

For the programming tasks: You can use Python, Java, or C++. None of your functions must be longer than TWENTY lines.

For all other tasks: Do not simply write down the final result. It should also be clear how you derived it.

Good luck!

This is the version of the exam with solution sketches. Do not distribute, it's for your personal use only. Don't use it when you do old exams for training, it's the worst way to learn. Use it only for checking your solutions, after you tried to solve the tasks yourself.

Task 1 (List intersection, 20 points)

In the following tasks, the two list lengths are denoted by k and n, ordered such that $k \leq n$.

1.1 (10 points) Write a function *intersect(list1, list2)* that takes two sorted lists and computes their intersection by binary searching each of the elements from the shorter list in the longer list. You must implement the binary search yourself. The running time must be $O(k \cdot \log n)$.

```
def intersect(list1, list2):
    if len(list1) > len(list2):
        list1, list2 = list2, list1
    result = []
    for x in list1:
        left, right = 0, len(list2)
        while left < right:
            mid = int((right - left)/2) + left
            if list2[mid] > x:
                right = mid
            elif list2[mid] < x:
                left = mid + 1
            else:
                result.append(x)
                break
    return result
```

1.2 (5 points) We have shown that the running time of the galloping search algorithm is $O(k \cdot \log(1+n/k))$. Prove that this running time is asymptotically never worse than that of the simple linear-time algorithm. You can use without proof that $e^x \geq 1+x$ for all real x.

1. We consider ln instead of log, the difference is only a constant factor
2. Because of the hint, $\ln(1+x) \leq x$ and hence $\ln(1+n/k) \leq n/k$
3. Therefore $k \cdot \ln(1+n/k) \leq k \cdot (1+n/k) = k+n$

1.3 (5 points) You are given two sorted lists and you want to compute their *union* as fast as possible. You care about the running time in *practice*, not in theory. How would you decide whether to use the simple linear-time algorithm or the galloping search algorithm? Explain the reasons for your choice.

1. If k/n is less than, say, 10 use galloping search, otherwise use zipper.
2. Explanation: galloping search is asymptotically always better, but it is a more complex algorithm with a larger constant factor. If the difference in ratios is large enough, it will pay off, otherwise zipper is faster. We saw this in the exercises.

Task 2 (Encodings and UTF-8, 20 points)

2.1 (5 points) Let us assume we want to encode the numbers 1, 2, 3, 4, 5, 6, 7, 8 and they all occur with equal probability. Specify a prefix-free entropy-optimal encoding and prove that it is entropy-optimal.

1. Encoding: $1 = 000$, $2 = 001$, $3 = 010$, $4 = 011$, $5 = 100$, $6 = 101$, $7 = 110$, $8 = 111$.
2. All code lengths are exactly 3, so the expected code length is also 3.
3. The entropy of the uniform distribution with eight probabilities is $-8 * 1/8 * \log_2(1/8) = 3$

2.2 (5 points) Shannon's source coding theorem says that there always is a prefix-free encoding with expected code length of at most $H(X) + 1$, where $H(X)$ is the entropy of the probability distribution of the symbols to be encoded. Under exactly which condition can we find a prefix-free encoding with expected code length of exactly $H(X)$, that is, without the $+1$? You should state the condition as simple as possible.

1. The source coding theorem finds an encoding with expected code length $\sum_i p_i \lceil \log_2 1/p_i \rceil$.
2. Without the rounding, this is *exactly* $H(X)$.
3. $\lceil \log_2 1/p_i \rceil = \log_2 1/p_i$ is true if and only if $\log_2 1/p_i$ is an integer.
4. That is true if and only if p_i is a negative power of 2 (for example: 1/2 or 1/4 or 1/1024).

2.3 (5 points) The Greek lowercase α has a UTF-8 codepoint of 945, while the greek uppercase A has a codepoint of 913. Write down the UTF-8 codes of the two characters in binary. Write them down one below the other.

```
Alpha: 11001110 10010001
alpha: 11001110 10110001
```

2.4 (5 points) Write a function *greek_lower_case(bytes)* that takes a UTF-8 encoded string (in the form of a byte array) that is composed entirely of Greek letters and converts all letters to lower case. You can assume that, just like in the ASCII code, the lowercase letters as well as the uppercase letters have consecutive codepoints. For example, β has codepoint 946, γ has codepoint 947, and so on.

Idea: for each two-byte character, simply set Bit#6 in the lowercase byte to 1

```
def greek_lower_case(bytes):
    for i in range(0, len(bytes)):
        if i % 2 == 1:
            bytes[i] |= 32  # 00100000b
    return bytes
```

Task 3 (Fuzzy prefix search, 20 points)

3.1 (5 points) Write down the 2-gram index for the following list of names: *abba*, *bab*, *aba*. Use padding from the left with a single $.

$a : 1, 3
$b : 2
ab : 1, 2, 3
ba : 1, 2, 3
bb : 1

3.2 (5 points) Let $x = aaab$. Use your 2-gram index from Task 3.1 to find all words y from the list from Task 3.1 such that $\text{PED}(x, y) \leq 1$. It is not enough if you just write down the result, it has to be clear how you used the 2-gram index. You can do the required PED calculations in your head and just write down the result (you don't have to write down the dynamic programming tables).

1. aaab $\rightarrow$ $aaab $\rightarrow$ $a and aa and aa and ab $\rightarrow$ 1, 3 and 1, 2, 3
2. The three words hence have the following numbers of 2-grams in common: 2, 1, 2
3. We need $\text{comm}(x', y') \geq |x| - 2 = 2$, hence only abba and aba remain
4. PED(aab, abba) = 2 $\rightarrow$ NO, PED(aab, aba) = 2 $\rightarrow$ NO

3.3 (10 points) Write a function *fuzzy_prefix_search(x, index)* that for a given string and a given 2-gram index returns the number of PED computations needed to compute all words y with $\text{PED}(x, y) \leq 1$ using the 2-gram index. Use a hash map for counting the number of matching 2-grams for each word. The running time should be linear in the total size of the inverted lists of the 2-grams of x.

```
def fuzzy_prefix_search(x, index):
    x = "$" + x
    counts = {}
    for i in range(0, len(x) - 1)
        qgram = x.substr(i, 2)
        list = index[qgram]
        for j in list:
            counts[list[j]] += 1
    num_peds_needed = 0
    for word_id in counts.keys():
        if counts[word_id] >= len(x) - 2:
            num_peds_needed += 1
    return num_peds_needed
```

Task 4 (Search web app, 20 points)

4.1 (10 points) Write a function *search(query, index, titles)* that for a query (given as an array of keywords), an inverted index (given as a map from words to arrays of pairs of document id and score), and the document titles (given as an arrays of strings), returns the title of the matching document with the highest score. A matching document does not necessarily have to contain all query words. Use a hash map to compute the various document scores. Do not assume that there is a function for merging sorted lists.

```
def search(query, index, titles):
    matches = {}
    for word in query:
        for id, score in index[word]:
            if id not in matches:
                matches[id] = 0
            matches[id] += score
    max_score, id_of_max = -1, 0
    for id, score in matches.items():
       if score > max_score:
           id_of_max = id
           max_score = score
    return titles[id_of_max]
```

4.2 (10 points) Write a function *search_server(socket, index, titles)* that for a given server socket and an index and document titles like in Task 4.1, listens to HTTP requests, interprets them as keyword queries and returns the best matching title using your function from Task 4.1. You can assume that *server_socket* has a method *accept* that returns a client socket, that the client socket has methods *receive* and *send* for reading and writing an arbitrary string in one slurp, that all requests are properly formed GET requests, and that no errors of any kind occur. You can also assume that all your titles contain only ASCII characters. The response should include header lines stating the status, the response length and the response type.

```
def search_server(server_socket, index, titles):
    while True:
        client = server_socket.accept()
        request = client.read()
        request = request[5 : request.find("HTTP")]
        query = request.split(" ")
        title = search(query, index, titles)
        headers = "Content-Type: text/plain\r\n" +
                  "Content-Length: " + len(title) + "\r\n" +
                  "\r\n"
        client.send(headers + title)
```

Task 5 (Latent Semantic Indexing, 20 points)

Consider the following term-document matrix (three terms, five documents):

$$A = \begin{pmatrix} 1 & 1 & 1 & 2 & 2 \\ 2 & 1 & 1 & 3 & 1 \\ 1 & 1 & 2 & 2 & 1 \end{pmatrix}$$

5.1 (5 points) Determine the rank of matrix A. Don't just write down the result, but also provide a proper argument.

1. The first, third, and fifth column are linearly independent (e.g., compute their determinant)
2. Hence the matrix has rank at least 3
3. A matrix with 3 rows can have row-rank at most 3
4. Since the rank is equal to the row-rank, the rank is exactly 3

5.2 (5 points) Change *at most two* entries of A such that its rank becomes 2. Change *at most four* entries of A such that its rank becomes 1. Don't just write down the result, but also provide a proper argument.

$$A = \begin{pmatrix} 1 & 1 & \mathbf{2} & 2 & 2 \\ 2 & 1 & 1 & 3 & 1 \\ 1 & 1 & 2 & 2 & \mathbf{2} \end{pmatrix} \qquad A = \begin{pmatrix} 1 & 1 & 1 & 2 & \mathbf{1} \\ \mathbf{1} & 1 & 1 & \mathbf{2} & 1 \\ 1 & 1 & \mathbf{1} & 2 & 1 \end{pmatrix}$$

5.3 (10 points) Write a function *is_row_orthonormal(matrix)* that determines if the product of the given matrix with its transpose is the identity matrix. Your function should not materialize the transpose and it should not use any special linear-algebra functions, just simple for-loops. You can assume that there are functions which give you the number of rows and columns of the matrix.

```
def is_row_orthonormal(matrix):
    for i in range(0, matrix.num_rows()):
        for j in range(0, matrix.num_rows()):
            sum = 0
            for k in range(0, matrix.num_columns()):
                sum += matrix[i][k] * matrix[j][k]
            if (i == j AND sum != 1) OR (i != j AND sum != 0):
                return False
    return True
```

Task 6 (SPARQL and POS-Tagging, 20 points)

6.1 (5 points) Write down the SPARQL query that produces a table of all female astronauts from Asia with the following columns: name of the astronaut, country of nationality, birth date. You can assume that you have a knowledge base with the appropriate entities and relations (the exact names of the relations are up to you). The body of your query should contain at least four triples and the corresponding subgraph should be connected.

```
SELECT ?person, ?nationality, ?date WHERE {
  ?person <profession> <Astronaut> .
  ?person <country of nationality> ?country .
  ?country <contained in> <Asia> .
  ?person <date of birth> ?date
}
```

6.2 (5 points) Write down the equivalent query in SQL, assuming that you have one two-column table for each of the relations you used in your query from Task 6.1.

```
SELECT profession.person, nationality.country, date_of_birth.date
FROM profession, nationality, contained_in, date_of_birth
WHERE profession.person = nationality.person
AND nationality.country = contained_in.country
AND date_of_birth.person = profession.person
AND contained_in.location = "Asia"
AND profession.profession = "Astronaut"
```

6.3 (10 points) Consider a HMM with two observable states *GOOD* and *BAD* (the mood) and two hidden states *SUNNY* and *RAINY* (the weather). Given a sequence $o_1, \ldots, o_n$ consider the following greedy algorithm to find a sequence $h_1, \ldots, h_n$ of hidden states: In the first step, find h_1 which maximizes $P(h_1) \cdot P(o_1|h_1)$. In subsequent steps, find h_{i+1} which maximizes $P(h_{i+1}|h_i) \cdot P(o_i|h_i)$. Find an example problem instance (all the probabilities and a sequence of observations) such that this greedy algorithm finds a different sequence of hidden states than the Viterbi algorithm (with explanation, of course). Hint: a sequence of two observations is enough.

1. Sequence of observations of length 2: GOOD, BAD
2. Pr(sunny) = 3/4, Pr(rainy) = 1/4
3. Pr(good|sunny) = 1, Pr(bad|sunny) = 0, Pr(good|rainy) = 1/2, Pr(bad|rainy) = 1/2
5. Pr(sunny|sunny) = 1, Pr(rainy|sunny) = 0, Pr(sunny|rainy) = 0, Pr(rainy|rainy) = 1
6. The greedy algorithm will pick SUNNY first, but then the overall probability is zero
7. The Viterbi algorithm will pick RAINY, RAINY with probability $1/4 \cdot 1/2 \cdot 1 \cdot 1/2 > 0$