Algorithmic Problem Solving Quick Test: Greedy Algorithms and Dynamic Programming

October 18th, 2019 - 45 minutes.
All documents forbidden.
One A4 handwritten sheet allowed.

Name: .....

The marking scheme is there to help you in discriminating the important questions from the easy ones and is subject to slight changes.

Longest increasing subsequence (LIS)

You are given an array X which contains n integers larger than zero. An increasing subsequence of X is a sequence of indices $i_0, \ldots, i_k \in \{0, \ldots, n-1\}$ for some $k \in \{0, \ldots, n-1\}$ such that $i_0 < i_1 < \cdots < i_k$ and $X[i_0] \leq X[i_1] \leq \cdots \leq X[i_k]$. For example in the array

$$X = [2, 4, 1, 3, 5, 7, 6, 3, 10, 8, 9]$$

the sequence $i_0 = 2, i_1 = 3, i_2 = 5, i_3 = 8$ is an increasing sub-sequence, because $X[2] = 1 \leq X[3] = 3 \leq X[5] = 7 \leq X[8] = 10$. You want to find an increasing subsequence in X that is as long as possible.

Task 1 (Greedy Algorithms)

a) Explain (shortly) one greedy strategy to find a (hopefully long) increasing subsequence. [2 pts]

b) What will this strategy compute for the sequence above? [1 pt]

c) How short is the sequence computed by your algorithm in the worst case? Give an example. [2 pts]

Task 2 (Dynamic Programming)

a) Consider the function $\mathrm{LIS}(i, ub)$ that finds the length of a longest increasing subsequence in the array X up to the index i and only contains values $\leq ub$ (*ub* is short for *upper bound*). Let max be the largest value in X. Write a recursive formula for the function LIS such that $\mathrm{LIS}(n-1, max)$ will contain the length of the longest increasing subsequence in X. Remember to mention the base cases. [4 pts]

b) Assume you want to store the results computed by the recursive formula inside a `cache`. Make a schematic drawing of your `cache` where you explain what the subproblems are and draw arrows of dependencies that indicate the subproblems needed to make a decision for the current problem. [4 pts]

c) Write a pseudo code for a dynamic program (recursive or sequential) that computes the length of the longest increasing subsequence. [5 pts]

d) What is the complexity of your algorithm? [2 pts]

1 a) Algorithm: Initialize the sequence as an empty list and add the index 0 to it. Iterate the array from left to right, starting at index 1, and compare the current value with the one at the index that was added last to the sequence. If it is larger, add the corresponding index to the sequence; otherwise, go to the next entry.

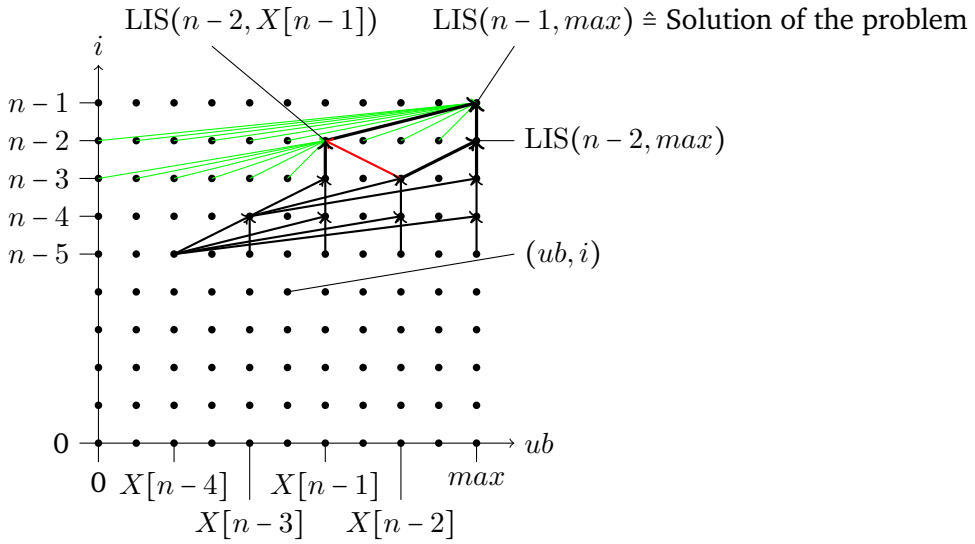
1 b) Sequence: $i_0 = 0, i_1 = 1, i_2 = 4, i_3 = 5, i_4 = 8$; $X[i_0] = 2 \leq X[i_1] = 4 \leq X[i_2] = 5 \leq X[i_3] = 7 \leq X[i_4] = 10$

1 c) In the worst case, the sequence generated by the greedy strategy will have length one while the longest sequence has length $n-1$.
Example: $X = [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]$.

2 a)

$$\text{LIS}(i, ub) = \begin{cases} 0 & \text{if } i = 0 \text{ and } X[i] > ub \\ 1 & \text{if } i = 0 \text{ and } X[i] \leq ub \\ \text{LIS}(i-1, ub) & \text{if } X[i] > ub \\ \max\{1 + \text{LIS}(i-1, X[i]), \text{LIS}(i-1, ub)\} & otherwise \end{cases}$$

2 b) The cache should have a field for every of the possible function calls of LIS. The value i is between 0 and $n-1$, while ub has a value between 0 and max, which is the largest value appearing in X. Hence, the cache should be a 2-dimensional array of size $n \times (max + 1)$.



In this figure, a point (ub, i) corresponds to the problem $\text{LIS}(i, ub)$ where we want the LIG between 0 and i containing only numbers $\leq ub$. The solution of the problem can be found at the point $(max, n-1)$ which corresponds to the problem $\text{LIS}(n-1, max)$. The green arrows represent the possible dependencies of $\text{LIS}(n-1, max)$ while the black arrows represent the actual dependencies, because only the subproblems $\text{LIS}(n-2, X[n-1])$ and $\text{LIS}(n-2, max)$ are relevant for the recursive formula. In this example $\text{LIS}(n-3, X[n-2])$ is not considered to calculate $\text{LIS}(n-2, X[n-1])$ (red arrow) since $X[n-1]$ is smaller than $X[n-2]$. However, it is considered to compute $\text{LIS}(n-2, max)$

2 c) recursive algorithm:

```
int cache[n][max+1] = {-1}; \\initialize array of size n times (max+1) with -1

int LIS(int i, int ub){
    int result;
    if(cache[i][ub] !=-1)
        return cache[i][ub];
    if(i=0 && X[i]<= ub)
        result 1;
    if(i=0 && X[i]> ub)
        result 0;
    if(X[i] > ub)
        result = LIS(i-1, ub);
    else
        result = max(LIS(i-1, ub), LIS(i-1, X[i]));
        cache[i][ub] = result;
        return result;
}
```

sequential version:

```
int LIS(int[] X){
```

```
   n = X.lenght();
   max = 0;
   for int i = 0 to n−1 do
       if (max < X[i])
           max = X[i];
   int cache[n][max+1] = {−1}; \\initialize array of size n times (max+1) with −1
   for int i = 0 to n−1 do
        for int ub = 0 to max do
          int result;
          if (i=0 && X[i]<= ub)
              result 1;
          if (i=0 && X[i]> ub)
              result 0;
          if (X[i] > ub)
              result = cache[i−1][ub];
          else
              result = max(cache[i−1][ub]), cache[i−1][X[i]]);
          cache[i][ub] = result;
   return cache[n−1][max];
}
```

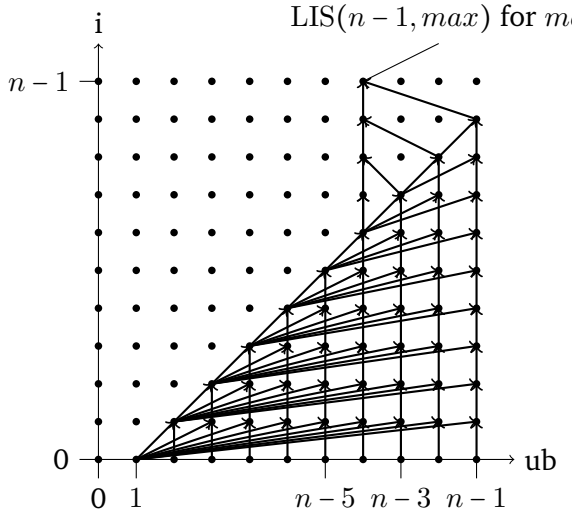
2 d) The cache needs to be initialized. Furthermore, we can compute each entry of the cache in $\mathcal{O}(1)$, because we only need to lookup two values. Hence, the complexity of the algorithm is $\mathcal{O}(n \cdot max)$, where max is the the largest entry in X.

Discussion (not expected, only for information) If max is very large, a complexity of $\mathcal{O}(n \cdot max)$ is problematic. Imagine $n = 10$ but $max = 1000000000$. If there are values inside the array which are in $\Omega(2^n)$, this complexity is no longer polynomial. This problem can be avoided by using the ***index*** of the largest value instead of the value itself:

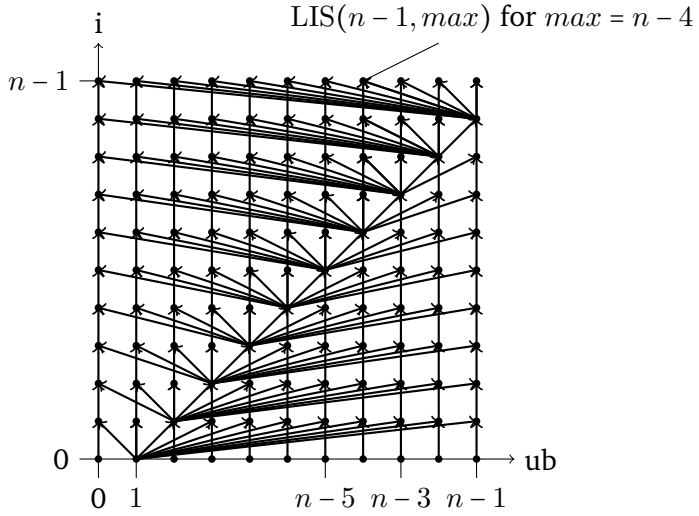
2 a)

$$\mathrm{LIS}(i, ub) = \begin{cases} 0 & \text{if } i = 0 \text{ and } X[i] > X[ub] \\ 1 & \text{if } i = 0 \text{ and } X[i] \leq X[ub] \\ \mathrm{LIS}(i-1, ub) & \text{if } X[i] > X[ub] \\ \max\{1 + \mathrm{LIS}(i-1, i), \mathrm{LIS}(i-1, ub)\} & otherwise \end{cases}$$

2 b) The cache should have a field for every of the possible function calls of LIS. The value i is between 0 and $n-1$, while ub has a value between 0 and $n-1$. Hence, the cache should be a 2-dimensional array of size $n \times n$.



In the above figure, only the (possible) dependencies for the recursive algorithm are presented.



In the above figure, all (possible) dependencies for the iterative algorithm are presented.

2 c) recursive algorithm:

```
int cache[n][n] = {-1}; \\initialize array of size n times n with -1

int LIS(int i, int ub){
    if(cache[i][ub] !=-1)
        return cache[i][ub];
    if(i=0 && X[i]<= X[ub])
        result 1;
    if(i=0 && X[i]> X[ub])
        result 0;
    if(X[i] > ub)
        result = LIS(i-1, ub);
    else
        result = max(LIS(i-1, ub), LIS(i-1, i));
    cache[i][ub] = result;
    return result;
}
```

sequential version:

```
int LIS(int[] X){
    n = X.lenght();
    int cache[n][n] = {-1}; \\initialize array of size n times n with -1
    for int i = 0 to n-1 do
        for int ub = 0 to n-1 do
            int result;
            if(i=0 && X[i]<= X[ub])
                result 1;
            if(i=0 && X[i]> X[ub])
                result 0;
            if(X[i] > X[ub])
                result = cache[i-1][ub];
            else
                result = max(cache[i-1][ub]), cache[i-1][i]);
            cache[i][ub] = result;
    max = 0;
    for int i = 1 to n-1 do
        if(X[max] < X[i])
            max = i;
    return cache[n-1][max];
}
```

2 d) The cache needs to be initialized. Furthermore, we can compute each entry of the cache in $\mathcal{O}(1)$, because we only need to lookup two values. Hence, the complexity of the algorithm is $\mathcal{O}(n \cdot n)$.