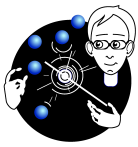


Runde 2T
Lösungen der Aufgaben

2T–1: Gerüchte, Lösung

Da die Aufgabenstellung einfach einen Prozess beschreibt, ist die Idee hier, den Ablauf der Kommunikation (effizient) zu simulieren.

Dazu führen wir *Ereignisse* ein, die eine bestimmte Bedeutung und einen bestimmten Zeitpunkt haben. Ein Ereignis kann weitere Ereignisse auslösen: Ereignis A kann Ereignis B auslösen, wenn A nicht nach B stattfindet. Nachdem ein einzelnes initiales Ereignis eingeführt wurde ist die Simulation folgender Prozess: Solange es noch Ereignisse abzuarbeiten gibt, arbeite das aktuelle Ereignis ab, welches am frühesten Zeitpunkt Stattfindet. Führe dabei alle Ereignisse ein, die vom aktuellen Ereignis ausgelöst werden.

(Eine solche Simulation führt zum korrekten Ergebnis, weil zu jedem Zeitpunkt alle Ereignisse, die zuvor abgearbeitet wurden auch vorher stattfanden, und es keine Ereignisse gibt die stattfanden, aber nicht abgearbeitet wurden, da während dem Abarbeiten nur neue Ereignisse eingeführt werden, die vom aktuellen Zeitpunkt gesehen in der Zukunft stattfinden.)

Für den Zweck dieser Aufgabe reicht es, einen einzelnen Typ von Ereignis einzuführen: Einen Zeitpunkt, wann eine Maus eine Nachricht an ihre Freunde sendet. Das initiale Ereignis beschreibt den Zeitpunkt $T = 0$, an dem Maus X ihre Nachrichten verschickt. Wenn das Ereignis "Maus i sendet die Nachricht" zum Zeitpunkt T_i zum ersten Mal stattfindet, dann müssen für alle Freunde j von Maus i Ereignisse "Maus j sendet die Nachricht" eingeführt werden, zum Zeitpunkt $T_j = T_i + S_j \cdot L$, wobei S_j die Lesegeschwindigkeit von Maus j und L die Länge der Nachricht ist.

Um darüber Buch zu führen, ob eine Maus die Nachricht schon einmal bekom-

men hat, reicht es eine einzelne Bool'sche Variable pro Maus einzuführen. Es ist ausserdem hilfreich die Anzahl der Mäuse zu zählen, die die Nachricht bereits gelesen/gesendet haben. In dem Moment, in dem dieser Zähler N erreicht, haben alle Mäuse die Nachricht gelesen.

Das einzige was nun noch klargestellt werden muss, ist wie Ereignisse eingeführt werden, und auf welche Art und Weise das aktuelle Ereignis bestimmt werden kann.

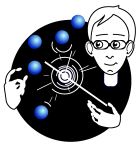
Die Laufzeit eines solchen Anzatzes ist dann beschränkt durch $\mathcal{O}(N + f(M) \cdot M)$, wobei $f(M)$ eine obere Schranke an die Laufzeit der Einführung eines neuen Ereignisses und der Bestimmung des neuen Ereignisses ist, und M die Gesamtzahl der Freundschaften ist.

Der Speicherverbrauch ist bei allen diskutierten Ansätzen $\mathcal{O}(N + M)$.

a) Da die Ereignisse der Reihe nach abgearbeitet werden, und der Zeitpunkt eines neuen Ereignisses immer konstant viel (nämlich L) später ist wie das des auslösenden Ereignisses, werden neue Ereignisse auch der Reihe nach eingeführt. Es reicht also, sich eine Schlange (Queue) von Ereignissen zu halten. Um ein Ereignis einzuführen, fügt man es hinten an die Schlange an, und das aktuelle Ereignis kann vorne von der Schlange entfernt werden. Da diese Operationen in konstanter Zeit ($\mathcal{O}(1)$) ausgeführt werden können, ist die Laufzeit des Algorithmus in $\mathcal{O}(N + M)$.

Dieser Algorithmus ist eine Version von Breitensuche.

b) Der naive Ansatz wäre, sich eine Liste aller eingeführten Ereignisse zu halten, und linear nach dem frühesten Ereignis zu suchen. Da diese Suche $\mathcal{O}(M)$ ist, würde der Algorithmus in $\mathcal{O}(N + M^2)$ laufen. Da nur das erste Ereignis einer Maus wichtig ist lässt sich dieser Ansatz noch verbessern: Anstatt einer Liste aller Ereignisse hält man sich ein Feld (Array), welches für jede der N Mäuse den Zeitpunkt des frühesten eingeführten Ereignisses festhält. Dadurch verbessert sich die Laufzeit zu $\mathcal{O}(N^2 + M)$, da jedes von N Ereignissen einmal



in $\mathcal{O}(N)$ gesucht wird, und höchstens M Ansätze gemacht werden ein Ereignis in $\mathcal{O}(1)$ einzuführen.

Ein besserer Ansatz ist es, die Ereignisse in einem Min-Heap abzulegen. Für die volle Punktzahl wurde verlangt, dass die Implementierung davon zumindest schematisch beschrieben wurde: Eine Art, so einen Min-Heap zu implementieren, ist, einen vollen (so voll wie möglich, es fehlen höchstens einige Blätter auf der untersten Stufe) gewurzelten Binärbaum mit den Elementen als Knoten zu verwenden, bei dem die Heap-Bedingung immer sichergestellt wird. Die Heap-Bedingung ist, das die Kinder grösser sind als ihre Eltern. Um ein neues Element einzuführen, fügt man es unten an den Baum an und vertauscht es dann so oft mit seinem Elternknoten, bis die Heap-Bedingung wieder erfüllt ist. Das Minimum ist einfach zu finden, es steht immer an der Wurzel des Baumes. Um es zu entfernen, vertauscht man es zuerst mit dem letzten Element im Baum, dann kann es einfach gestrichen werden. Die Heap-Bedingung kann wiederhergestellt werden, indem man das andere Element so lange mit seinem kleineren Kind tauscht, bis es kleiner ist als alle seine Kinder. Da alle diese Operationen nur konstant viele Operationen pro Stufe des Baumes ausführen, ist ihre Laufzeit in $\mathcal{O}(\log M)$.

Die Laufzeit des Algorithmus wird damit $\mathcal{O}(N + M \log M) \subseteq \mathcal{O}(N + M \log N)$.

(Dieser Algorithmus kann analog dazu wie der naive Ansatz verbessert wurde verbessert werden, indem eine Datenstruktur verwendet wird, die es in (amortisiert) $\mathcal{O}(1)$ erlaubt, einen Wert, der bereits eingefügt wurde zu verkleinern. Dadurch verbessert sich die Laufzeit des Algorithmus zu $\mathcal{O}(N \log N + M)$. Der Fibonacci-Heap ist eine Datenstruktur mit diesen Laufzeiten. Diese asymptotische Optimierung war für die Vollpunktzahl nicht notwendig.)

Dieser Algorithmus ist eine Version von Dijkstras Algorithmus für die Suche nach kürzesten Wegen.

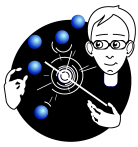
c) Fast dieselbe Idee funktioniert immer noch: Jetzt gehört zu jedem Ereignis einfach noch entweder die Nachricht X oder Y, wobei Events, die Nachricht Y enthalten, früher abgearbeitet werden als gleichzeitige Ereignisse des anderen Typs. Man behält sich ausserdem für jede Maus einen Zeitpunkt F_i, an dem sie alle Nachrichten fertig gelesen hat, der am Anfang auf 0 gesetzt ist. Der

Zeitpunkt, wo der Event für Maus j eingeführt wird ist dann $T_j = \max(T_i, F_i) + S_j \cdot L$, wobei L die Länge der entsprechenden Nachricht ist (unterschiedlich lange Nachrichten X und Y breiten sich ja jetzt unterschiedlich schnell aus.)

Implementierungen

a)

```
#include <vector>
#include <queue>
using namespace std;

struct Mouse{
  int speed;
  bool read = false;
  vector<int> friends;
};

struct Event{
  int mouse;
  int time;
};

int main(){
  // IO //
  int n,m; // n -- #mice; m -- #friendships;
  int x,l; // x -- Mouse X; l -- length of gossip
  vector<Mouse> mice(n); // network
  int answer;
  ////////
  queue<Event> q;
  q.push({x,0});
  int n_readers = 0;
  while(!q.empty()){
    Event current = q.front();
    q.pop();
    int i = current.mouse;
    int t = current.time;
    if(mice[i].read) continue;
    mice[i].read = true;
    n_readers += 1;
    if(n_readers == n){
```

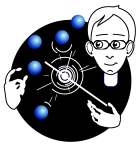


```
35        answer = t;
36        break;
37      }
38      for(int j : mice[i].friends){
39        q.push({j, t + 1});
40      }
41    }
42  }
```

b)

```
1  #include <vector>
2  #include <algorithm>
3  using namespace std;
4
5  #define LT_BY(T,F) bool operator<(const T& other)const{ return F<
       other.F; }
6
7  struct Mouse{
8    int speed;
9    bool read = false;
10   vector<int> friends;
11 };
12
13 struct Event{
14   int mouse;
15   int time;
16   LT_BY(Event,time)
17 };
18
19 struct Heap{
20   // we represent the binary tree as a vector.
21   // the children of node i are 2*i+1 and 2*i+2.
22   vector<Event> events;
23   bool empty(){ return events.size() == 0; }
24   void push(Event event){
25     events.push_back(event);
26     for(int i=(int)events.size()-1;i;i=(i-1)/2){
27       if(events[i] < events[(i-1)/2]){
28         swap(events[i], events[(i-1)/2]);
29       }
30     }
31   }
32   Event pop(){
```

```
33      int n = (int)events.size();
34      Event r = events[0];
35      swap(events[0],events[n-1]);
36      events.pop_back(), n -= 1;
37      int i = 0, j = 0;
38      while(2*i+1<n){
39        int j = 2*i+2>=n || events[2*i+1] < events[2*i+2] ? 1 : 2;
40        if(events[2*i+j] < events[i]){
41          swap(events[2*i+j], events[i]);
42          i = 2*i+j;
43        }
44      }
45      return r;
46    }
47  };
48
49  int main(){
50    // IO //
51    int n,m; // n -- #mice; m -- #friendships;
52    int x,l; // x -- Mouse X; l -- length of gossip
53    vector<Mouse> mice(n); // network
54    int answer;
55    ////////
56    Heap q;
57    q.push({x,0});
58    int n_readers = 0;
59    while(!q.empty()){
60      Event current = q.pop();
61      int i = current.mouse;
62      int t = current.time;
63      if(mice[i].read) continue;
64      mice[i].read = true;
65      n_readers += 1;
66      if(n_readers == n){
67        answer = t;
68        break;
69      }
70      for(int j : mice[i].friends){
71        int s = mice[j].speed;
72        q.push({j, t + s*l});
73      }
74    }
75  }
```



c)

```
#include <vector>
#include <algorithm>
#include <utility>
using namespace std;

struct Mouse{
  int speed;
  bool read[2] = {false, false}; // !
  int busy_until = 0; // !
  vector<int> friends;
};

struct Event{
  int which;
  int mouse;
  int time;

  bool operator<(const Event& other)const{ // !
    return make_pair(time, 1-which) <
      make_pair(other.time, 1-other.which);
  }
};

struct Heap{ /* ... */ };

int main(){
  // IO //
  int n,m; // n -- #mice; m -- #friendships;
  int x,y,l[2]; // x,y -- Mouse X/Y; l[0;1] -- length of gossip
  vector<Mouse> mice(n); // network
  int answer;
  ////////
  Heap q;
  q.push({0,x,0}), q.push({1,y,0}); // !
  int n_readers = 0;
  while(!q.empty()){
    Event current = q.pop();
    int w = current.which; // !
    int i = current.mouse;
    int t = current.time;
    if(mice[i].read[w]) continue; // !
    mice[i].read[w] = true; // !
    n_readers += 1;
    if(n_readers == 2*n){ // !
      answer = t;
```

```
46        break;
47      }
48      for(int j : mice[i].friends){
49        int s = mice[j].speed;
50        int b = mice[j].busy_until; // !
51        b = max(b,t) + s*l[w]; // !
52        mice[j].busy_until = b; // !
53        q.push({j,b});
54      }
55    }
56  }
```

Lösungen der Beispiele

a) Es dauert $4 \cdot 26 = 104$ Sekunden.

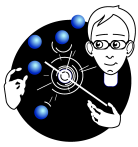
b) Es dauert $15 \cdot 15 = 225$ Sekunden.

2T–2: Abspülen, Lösung

Wie lösen die Aufgaben folgendermassen: Zu jedem Zeitpunkt vergleichen wir den obersten Teller auf dem dreckigen Stapel mit dem obersten Teller im Waschbecken. Im Allgemeinen gibt es zwei Fälle:

- Der Teller auf dem dreckigen Stapel ist grösser ⇒ wir legen diesen in das Waschbecken.
- Der Teller im Waschbecken ist grösser ⇒ wir legen diesen in den Schrank.

Die Spezialfälle, in denen wir keine Wahl haben:



- Das Waschbecken ist leer, es hat aber noch dreckige Teller (z.B. der Anfangszustand) ⇒ Wir legen den obersten Teller ins Waschbecken
- Das Waschbecken ist leer und es hat keine dreckigen Teller mehr ⇒ Wir geben 'YES' aus.
- Es hat keine dreckigen Teller mehr und wir können noch Teller in den Schrank legen ⇒ Wir legen den einen Teller in den Schrank
- Es hat keine dreckigen Teller mehr und wir können keinen Teller mehr in den Schrank legen ⇒ Wir geben 'No' aus.

Beweis Wir müssen zeigen, dass wir mit obiger Strategie keinen Fehler machen. Der einzige Moment, in dem wir einen Wahl haben, ist in den allgemeinen Fällen, wenn wir die Grösse der Teller vergleichen.

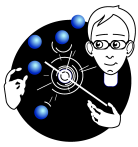
- Der Teller auf dem dreckigen Stapel ist grösser: Wenn wir in diesem Moment den sauberen (den kleineren) Teller in den Schrank stellen würden, würden wir irgendwann ein Problem haben, da über diesem Teller im Schrank nur kleinere Teller stehen dürfen. Wir haben aber immer noch den dreckigen (den grösseren) Teller.
- Der Teller im Waschbecken ist grösser: Wenn wir nun vom dreckigen Stapel den kleineren Teller draufstellen, werden wir wieder irgendwann ein Problem haben. Denn aus dem Waschbecken müssen wir zuerst den kleineren Teller in den Schrank stellen, bevor wir den grösseren Teller erreichen. Diesen können wir aber dann gar nicht mehr in den Schrank stellen, da dort schon ein kleinerer hereingestellt wurde.

Wir haben also gezeigt, dass wir mit unserem Algorithmus sicher nicht die falsche Wahl treffen. Damit ist unser Algorithmus korrekt; er wird den Endzustand erreichen, falls dieser erreichbar ist.

Kostenanalyse Jeder Schritt benötigt konstante Zeit, die Laufzeit und damit auch der Speicherverbrauch sind deshalb $\mathcal{O}(n)$.

Implementierung

```
#include <iostream>
#include <stack>
#include <vector>

using namespace std;

int n;
double tmp;
std::vector<double> dirty;
std::stack<double> cupboard;
std::stack<double> washbasin;

int main() {
  cin >> n;

  for(int i=0; i<n; i++){
    cin >> tmp;
    dirty.push_back(tmp);
  }
  reverse(dirty.begin(), dirty.end());

  while(dirty.size() > 0){
    if(washbasin.empty() || dirty.back() > washbasin.top()){
      tmp = dirty.back();
      dirty.pop_back();
      washbasin.push(tmp);
    }
    else{
      if(cupboard.empty() || cupboard.top() > washbasin.top()){
        tmp=washbasin.top();
        washbasin.pop();
        cupboard.push(tmp);
      }
      else{
        cout << "NO";
        return 0;
      }
    }
  }
  while(washbasin.size() > 0){
    if(cupboard.top() > washbasin.top()){
      tmp = washbasin.top();
      washbasin.pop();
      cupboard.push(tmp);
    }
```



```
46      else{
47        cout << "NO";
48        return 0;
49      }
50    }
51    cout << "YES";
52
53  }
```

2T–3: Käseversteck, Lösung

Teilaufgabe a)

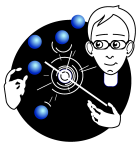
Der Käse muss beim Blatt 5 sein.

Teilaufgabe b)

Wir erhalten das richtige Blatt folgendermassen: Wir stellen n in Binärform dar. Wir wollen dabei $h-1$ Stellen haben, müssen also vorne eventuell noch Nullen anfügen. Dann spiegeln wir die ganze Bitfolge und geben die so entstandene Zahl in normaler Dezimalform aus.
Für $h = 6$ und $n = 13$ würde dies also so aussehen:

$$12_{10} = 1101_2 = 01101_2 \Rightarrow 10110_2 = 22_{10}$$

Beweis Wir können dies induktiv zeigen. Für $h = 1$ ist die Aussage trivial. Wir nehmen nun an, dass wir die Aussage für $h-1$ bewiesen haben und wollen zeigen, dass sie auch für h gilt:
Wir erkennen, dass jede Maus mit einem geraden n nach links geht und jede mit ungeradem n nach rechts.
Betrachten wir die "linken Mäuse": Diese sind nun in einem Baum der Höhe $h-1$ und wir wissen deshalb, dass sie bei dem Blatt ankommen, das unser Algorithmus für $h-1$ berechnet. Da diese Mäuse aber ein gerades n haben, ist ihre letzte Ziffer in Binärdarstellung eine 0 und diese würde in unserem Algorithmus keinen Unterschied machen. Daher stimmt unser Algorithmus auch für h.
Für die "rechten Mäuse": Diese sind ebenfalls in einem Baum der Höhe $h-1$, wobei aber die Nummerierung der Blätter bei 2^{h-2} beginnt. Unser Algorithmus für $h-1$ berechnet also eine Zahl, die um 2^{h-2} zu klein ist. Da diese Mäuse aber ein ungerades n haben, sieht der Algorithmus für h noch eine zusätzliche



1 zuvorderst (an der Stelle $h-1$). Dies entspricht gerade einer Addition von 2^{h-2}

Implementierung Wir können dies in $\mathcal{O}(\log(n))$ implementieren. Betrachte dabei, dass $\lfloor \log_2(n)+1 \rfloor$ die Anzahl Bits von n sind.

```
1  int result=0;
2  int temp;
3  int bit;
4
5  for(int i=0; i<int(log2(n)+1); i++){
6      temp = n&(1<<i); //set all bits to 0 except at the i-th
           position
7      temp = temp >> i; //shift the 1 all to the right
8      temp = temp << h-1-i; //shift the bit to the mirrored place
9      result += temp; //add the number to the result
10 }
```

Teilaufgabe c)

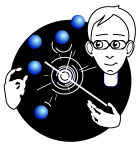
Nach 2^{h-1} Mäusen zeigen wieder alle Wegweiser nach links, die Situation ist also wieder gleich wie am Anfang; für $n = 2^{h-1}$ kommt die Mause am gleichen Ort an, wie für $n = 0$. Die nächste Maus folgt dem gleichen Weg wie für $n = 1$ etc. Wir können also zuerst $n \mod 2^{h-1}$ berechnen und dann den gleichen Algorithmus anwenden.

2T–4: Internat, Lösung

Im folgenden werden wir Bücher, die mindestens einmal herausgenommen und an einem anderen Ort hingelegt wurden, als "bewegt" bezeichnen. (Wir können den Fall, in dem ein Buch wieder an denselben Ort zurückgelegt wird, ignorieren, da wir die *Mindest*anzahl bewegter Bücher bestimmen wollen.)

Die wichtigste Beobachtung für diese Aufgabe ist, dass zu jedem Zeitpunkt die Teilfolge der (noch) nicht bewegten Bücher aufsteigend ist. Dessen kann man sich wie folgt vergewissern: Sei S die Teilfolge der nicht bewegten Bücher. Am Anfang wurde noch keines der Bücher bewegt, also umfasst S alle Bücher und nach Annahme sind sie aufsteigend sortiert. Wenn ein Buch bewegt wird, bleibt S entweder gleich, nämlich dann wenn es sich um ein bereits bewegtes Buch handelt, oder das Buch wird aus S entfernt. Da Bücher aus S per Definition nicht bewegt wurden, kann sich ihre Reihenfolge untereinander bei keinem Schritt ändern; S bleibt also auch nach einer beliebigen Anzahl von Schritten aufsteigend. (Diese Art von Argumentation nennt man übrigens Induktion).

Die Anzahl bewegter Bücher ist durch $n - |S|$ gegeben; die Mindestanzahl zu finden ist gleichbedeuted damit, $|S|$ zu maximieren. Da S eine aufsteigende Teilfolge ist, bekommen wir eine untere Schranke für $|S|$ indem wir die (eindeutige) Länge der längstmöglichen aufsteigenden Teilfolgen (LAT) in der Eingabefolge bestimmen. Doch kann jede aufsteigende Teilfolge S sein? In Teilaufgabe c) ist das der Fall, denn jedes Buch kann an eine beliebige Position verschoben werden. In der Teilaufgabe b) muss S zusammenhängend sein; durch das Verschieben von Büchern an den Anfang oder ans Ende kann man nämlich nie ein Buch zwischen zwei nicht bewegte Bücher bewegen. S ist in diesem Fall also ein aufsteigender Substring (ein deutschsprachiges Äquivalent zu diesem Wort scheint nicht zu existieren), und wiederum kann man sich recht leicht überzeugen, dass jeder aufsteigende Substring S sein kann. Bei der Teilaufgabe a) kann es kein bewegtes Buch rechts von S geben, da man Bücher nur an den Anfang bewegen kann. S ist also das längstmögliche aufsteigende Suffix (in diesem Fall kann es nur eines geben), und wiederum ist es auf jeden Fall möglich, dieses S durch erlaubte Schritte zu erzeugen.



Implementierung und Laufzeitanalyse

Sei im folgenden für $1 \leq k \leq n$ `B[k]` die Nummer des k-ten Buchs der Eingabefolge. Wir werden für die Besprechung der ersten beiden Teilaufgaben eine Zählvariable `c` verwenden, die anfangs jeweils mit `1` initialisiert wird.

a) Beachte, dass das letzte Buch (`B[n]`) auf jeden Fall im längsten aufsteigenden Suffix enthalten ist. Wir setzen `c = 1` und wiederholen folgenden Schritt:

- Falls `c < n` und `B[n - c] < B[n - c + 1]`, inkrementiere `c` um 1.
- Ansonsten, breche ab und gebe `n-c` aus.

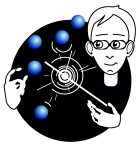
Die Bücher `B[n - c + 1]`, ..., `B[n]` bilden ein aufsteigendes Suffix, das man nicht erweitern kann (da `B[n - c] > B[n - c + 1]` oder `c == n`) und somit also maximal sein muss. Da jedes Buch maximal einmal angeschaut wird, ist die Laufzeit in $O(n)$.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

size_t li_suffix(unsigned int *B, size_t n){
  size_t c = 1;
  while (c < n && B[n-c] < B[n-c+1])
    c++;
  return c;
}

int main(void){
  size_t n, k;
  unsigned int *B;

  scanf("%zu ", &n);
  B = (unsigned int *)malloc((n+1)*sizeof(unsigned int));
  for (k = 1; k <= n; k++)
    scanf("%u ", &B[k]);

  printf("%zu\n", li_suffix(B, n));
  free(B);
  return 0;
}
```

b) Um den längsten aufsteigenden Substring zu finden, kann man die Lösung der vorigen Teilaufgabe rekursiv anwenden. Sei wiederum c die Länge des längsten aufsteigenden Suffixes. Es gibt zwei mögliche Fälle: Entweder ist dieses Suffix der längste aufsteigende Substring oder die beiden sind disjunkt (d.h. es gibt kein Buch, das in beiden enthalten ist). Überlege dir als Übung, welches die anderen Fälle sind und wieso sie nicht auftreten können. Im zweiten Fall ist der l.a. Substring ganz in der Folge enthalten, die übrigbleibt, wenn man den l.a. Substring entfernt (in der Notation der vorherigen Teilaufgabe ist dies $\texttt{B}[\texttt{1}], \ldots, \texttt{B}[\texttt{n} - \texttt{c}]$); seine Länge berechnen wir mittels eines Rekursionsschritts. Abschliessend wird `n` minus die grössere der Längen von l.a. Suffix und l.a. Substring von $\texttt{B}[\texttt{1}], \ldots, \texttt{B}[\texttt{n} - \texttt{c}]$ ausgegeben.

Dieses Mal wird jedes Buch genau einmal angeschaut; das ergibt eine Laufzeit in $O(n)$.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

size_t li_suffix(unsigned int *B, size_t n){
  size_t c = 1;
  while (c < n && B[n-c] < B[n-c+1])
    c++;
  return c;
}

int main(void){
  size_t n, k, c, maximum;
  unsigned int *B;

  scanf("%zu ", &n);
  B = (unsigned int *)malloc((n+1)*sizeof(unsigned int));
  for (k = 1; k <= n; k++)
    scanf("%u ", &B[k]);

  maximum = 0;
  k = n;
  while (k > 0){
    c = li_suffix(B, k);
    maximum = (c > maximum) ? c : maximum;
```



```
26      k -= c;
27    }
28
29    printf("%zu\n", n-maximum);
30    free(B);
31    return 0;
32  }
```

c) Für die Bestimmung der Länge der längsten aufsteigenden Teilfolge gibt es einen Standardalgorithmus, der z.B. in `https://en.wikipedia.org/wiki/Longest_increasing_subsequence` beschrieben wird. Dieser läuft in $O(n \log n)$. Für diese Aufgabe gibt aber auch ein Algorithmus mit Laufzeit $O(n^2)$ den Grossteil der Punkte.

Lösungen der Beispiele

a) Das längste aufsteigende Suffix ist `1457`, die Mindestanzahl bewegter Bücher also 4. Eine mögliche Schrittabfolge:

1. `3` an den Anfang.
2. `2` an den Anfang.
3. `6` an den Anfang.

b) Der einzige längstmögliche aufsteigende Substring ist `347`, die Mindestanzahl bewegter Bücher also 3. Eine mögliche Schrittabfolge:

1. `5` an den Anfang.
2. `2` an den Anfang.
3. `1` ans Ende.

4. `6` ans Ende.

c) Eine LAT ist `1257`, die Mindestanzahl bewegter Bücher also 3. Eine mögliche Schrittabfolge:

1. `3` an den Anfang.
2. `4` ans Ende.
3. `6` zwischen 2 und 5.