Training für Programmierwettbewerbe - Exam Questions

Hanno Barschel (174761)

1 Introduction

(1) Zeit Komplexität von drei Funktionen:

- **foo(.)** hat eine Zeitkomplexität von $\mathcal{O}(n)$ (wobei n die Länge des gegebenen Vektors ist) und eine Platzkomplexität von $\mathcal{O}(n)$.

  Das folgt direkt daraus, dass wir genau 2 Vorschleifen die jeweils den gesamten Vektor durchgehen haben also $\mathcal{O}(2n) = \mathcal{O}(n)$.

- **print_ pairs(.)** hat eine Zeitkomplexität von $\mathcal{O}(n^2)$ und eine Platzkomplexität von $\mathcal{O}(n)$.

  Für jedes Element des Vektors ruft die Funktion nochmal jedes Element des Vektors auf und printed beide als Paar aus. Somit kommt eine Zeitkomplexität von $\mathcal{O}(n^2)$ zustande.

- **print_ unordered_ pairs(.)** hat eine Laufzeitkomplexität von $\mathcal{O}(n^2)$ und eine Platzkomplexität von $\mathcal{O}(n)$.

  Für jedes Element des Vektors rufen wir alle Folgeelemente auf. Das sind es $n - (i + 1)$ Aufrufe für Element i (i startet bei 0). Damit kommen wir auf eine Laufzeit von

  $$\sum_{i=0}^{n-1}(n - (i + 1)) = n^2 - \sum_{i=1}^{n} i = n^2 - \frac{n \cdot (n + 1)}{2} = n^2 - \frac{n^2}{2} - \frac{1}{2} \in \mathcal{O}(n^2).$$

- **all_ fib(.)** hat eine Zeitkomplexität von $\mathcal{O}(n)$ und eine Platzkomplexität von $\mathcal{O}(n)$.

  Die for-Schleife printed uns im Prinzip alle Fibonacci Zahlen bis n aus. Sobald wir eine Null in $memo[i]$ finden sind aber bereits $memo[i-1]$ und $memo[i-2]$ berechnet, somit müssen wir diese nur addieren und haben keine wirklich tiefe Rekursion.

(2) Algorithmen Beispiele:

- $\mathcal{O}(1)$: Für eine gegebene Hashtabelle und einen Index gibt der Algorithmus den Wert für den Index zurück.
- $\mathcal{O}(\log(n))$: In einer sortierten Liste ein gegebenes Element suchen.
- $\mathcal{O}(n)$: Der Algorithmus berechnet zu einem gegeben Array die Summe aller Elemente und gibt diese zurück.
- $\mathcal{O}(n \cdot \log(n))$: Ein guter Sortier Algorithmus (Merge Sort).
- $\mathcal{O}(n^2)$: Quicksort (worst-case Analyse).

(3) Ersteinmal natürlich ordentlich zuhören wenn die Frage gestellt ist. Dabei sollten wir uns alles merken und auch auf Besonderheiten achten. Dann wählen wir ein geeignetes Beispiel. Dabei sollten wir immer im Hintergrund behalten, dass oft Spezialfälle besonders zu betrachten sind. Anschließend schreiben wir einen brute force Algorithmus auf und betrachten dessen Zeit sowie Platzkomplexität. Jetzt optimieren wir diesen noch indem wir das Ganze möglichst einfach machen, überflüssige und doppelte Arbeit vermeiden. Mit diesen Betrachtungen verbessern wir den Brutforce Algorithmus schließlich noch.

(4) Gegeben sei eine Liste mit Natürlichen Zahlen. Gesucht sei die maximale Differenz zweier Werte im Array wobei der vordere (steht weiter vorne im Array) Wert kleiner als der hintere sein muss. Wir iterieren mittels zweier indexe, tmp_min und tmp die am Anfang beide auf 0 gesetzt werden. Gleichzeitig haben wir die Variable $max\_diff = 0$ in der die bisher gefundene maximale Differenz gespeichert wird.

Nun beginnen wir mittels einer for-Schleife mit Variablen tmp über den Array zu iterieren: Ist der Wert im Array an Stelle tmp kleiner oder gleich als der an der Stelle tmp_min so setzen wir tmp_min auf tmp. Ist tmp größer so berechnen wir die Differenz zwischen tmp und tmp_min. Ist diese größer als max_diff so ersetzen wir diese, ansonsten inkrementieren wir tmp einfach weiter.

(5) Ich würde es mit einer Hashmap checken: Wir testen erst einmal ob die Strings die gleiche Länge haben (wenn nicht dann return 0). Danach erstellen wir 2 Hashmaps (für jedes Wort eine). Anschließend gehen wir mittels einer for-Schleife beide strings durch und für jeden Buchstaben inkrementieren wir den zugehörigen Hashwert. Abschließend vergleichen wir beide Hashmaps.

$return(hashmap\_string1 == hashmap\_string2)$.

2 Arrays and Bits

(1) Für geordnete Mengen, wo jeder Eintrag einfach nur angibt ob das Element in der Menge ist oder nicht, können wir einfach die Bitoperationen anwenden um beispielsweise leicht die Anzahl an Elementen herausfinden die in beiden Mengen enthalten sind (durch den & Operator und danach rufen wir die count-Funktion auf). Zusätzlich können wir neue Mengen erstellen die...

- ...alle Elemente enthalten die mindestens in einer der beiden Listen enthalten sind.
- ...alle Elemente enthalten die nur in genau einer der beiden Listen enthalten sind.

Auch können wir sehr leicht das Komplement, Vereinigung, Durchschnitt und Differenz bilden. Natürlich geht noch viel mehr. Bitoperationen haben den Vorteil, dass sie deutlich schneller sind, als über einen Array zu iterieren.

(2) Negative vorzeichenbehaftete Integer haben ein Vorzeichenbit, dass auf 1 gesetzt ist. Ein logischer Rechts shift würde jetzt einfach alle bits nach rechts schieben und in die höchste Stelle eine 1. Somit wäre das Resultat dann plötzlich positiv.

Ein arithmetischer Rechtsshift lässt das erste Bit auf 1 und ersetzt die Bits nach dem Vorzeichenbit mit 1 (so viele wie geschoben wird). Damit bleibt die Zahl negativ und die Operation entspricht (etwa) einer Division mit 2, was unserer Vorstellung von einem Rechtsshift entspricht.

(3) $(n\&(n-1)) == 0$ testet ob die bitdarstellung von n und $n-1$ 0 ergeben. Das ist der Fall, falls $n = 0$, $n = 1$ ist (&0 ergibt immer 0). Angenommen n ist eine 2er Potenz (also an k.ter Stelle eine 1 im bitstring). Dann ist in n-1 genau jede 0 die vor der k.ten Stelle kommt auf 1 gesetzt und die 1 in der k.ten Stelle wird zu 0. In den höheren Stellen ändert sich nichts. Somit ergibt für diesen Fall die Formel auch 0.

Angenommen n ist nicht durch 2 teilbar. In diesem Fall werden in n-1 alle Nullen hinter der ersten 1 in n auf 1 gesetzt und die erste 1 auf 0. Nach der Annahme, dass n keine 2er Potenz ist, existiert jetzt ein höheres bit, dass sowohl in n als auch in n-1 auf 1 gesetzt ist. Somit ergibt die Formel nicht 0.

(4) Wir wollen die Anzahl der Einserbits in der Variablen sum speichern.

Wir gehen in einem for loop von 0 bis sizeof(integer) über jeden einzelnen bit indem wir die gegebene Zahl n immer um 1 shiften und dann $n\&1$ auf unsere Summe addieren. $n\&1$ ist genau gleich dem letzten bit und so ergibt der Algorithmus die geforderte Summe.

(5) Wir wollen den reversed Integer in result speichern.

Um die Bits in einem gegeben Integer umzudrehen iterieren wir mittels eines for-loops über alle bits (siehe (4)), shiften die gegebene Zahl immer um 1 und addieren das Ergebnis $n\&1$ auf result drauf. Anschließend shiften wir result einmal nach Links.

(6) Die Quicksortpartitionierungsfunktion gibt für eine übergebene Liste, eine Liste aus wo das Pivotelement bereits an der richtigen Stelle steht. Nun gehen wir die Liste durch und testen für jedes Element ob es größer als das Pivotelement ist. Ist das der Fall so tauschen wir es nach hinten.

Wenn wir Einträge nach gerade und ungerade sortieren wollen können wir ähnlich vorgehen nur, dass wir wir testen ob das Element gerade ist. Ist das der Fall so tauschen wir es nach hinten.

3 Arrays and Strings

(1) Als erstes testen wir ob das erste Zeichen ein $'-'$ ist, ist das der Fall so starten wir die for-Schleife Index 1 ansonsten bei Index 0. Wir initialisieren einen un-

signed Integer mit 0 und starten die besagte for-Schleife. Diese geht von einem Index (entweder 0 oder 1) bis zur Länge des Strings alle string Teile durch. Für jeden Durchgang multiplizieren wir *result* mit 10 um alle bisherigen Zahlen um 1 nach links zu verschieben. Jeder Charakter wird mittels $s[i] -' 0'$ zu einem Integer zwischen 0 und 9 konvertiert und auf *result* addiert.

(2) Um auf die einzelnen Dezimalziffern unseres Integers zuzugreifen verwenden wir den modulo Operator. Solange die gegebene Nummer nicht 0 ist rechen wir modulo 10, bekommen so die letzte Ziffer, wandeln diese mit $char(.) +' 0'$ in einen char um, der in Ascii genau der Ziffer entspricht und fügen diesen an unseren String an. Anschließend teilen wir die Nummer durch 10. Jetzt haben wir allerdings immer nur die kleinste Ziffer genommen also müssen wir noch den string reversen.

(3) ASCII hat 128, was relativ wenig sind, aber für viele Texte reicht es und somit ist die Kodierung optimal, weil wir pro Buchstabe nur 8 bit brauchen (bzw 7 aber das wird meist dann auch in 8 gespeichert).

EXTENDED ASCII ist ähnlich wie ascii nur haben wir das doppelte an Bits (256) also brauchen wir zur Speicherung genau 8 bits.

UTF-8 ist eine dynamische Kodierung, d.h. wir haben einen Präfix der uns angibt wie viele bytes wir für einen Charakter lesen müssen. Damit können wir sowohl normales Ascii umsetzen als auch irgendwelche krassen Sonderzeichen in mehrere Bytes kodieren.

UTF-16 nutzt 16 oder 32 bits und hat somit auch dynamische Länge. Für normales Ascii bietet sich nicht an, weil wir dann immer 16 bits hätten wovon wir aber nur 7 benötigen.

UTF-32 nutzt 32 bit zur Speicherung und ist somit von fester Länge. Hier brauchen wir keinen Präfix, weil wir feste Länge haben, also nutzen wir unseren Platz besser aus. Andererseits wäre diese Kodierung eine Platzverschwendung für Texte mit Ascii Kodierung.

(4) Um einen String so schnell wie möglich zu säubern nutze so viel wie möglich inplace Replacemenzt, wenn das Wort kürzer ist, und lösche am Ende redundante Lücken. Falls du Regex Ausdrücke verwendest versuche diese für mehr Geschwindigkeit und Effiziens hard zu coden. Nutze so wenig Fallunterscheidungen wie möglich.

(5) Die regex Klasse macht viele Fallunterscheidungen und ist somit relativ langsam. nicht inplace?.

(6) Run-length Encoding wird für Daten genommen, die viele sich wiederholende Charakters haben. Bei normalen Strings ist das meistens nicht der Fall also eher nicht. Eher dort wo immer ähnliche Muster auftreten oder es Blöcke, die nur einen Buchstaben haben gibt.

4 LinkedLists und so

(1) Sei eine verlinkte Liste und ein Element daraus gegeben, dass wir in konstanter Zeit löschen wollen.

Das geht indem wir die Daten aus dem Nachfolger in unseren Knoten schreiben und den pointer auf die übernächste Node (node->next->next) zeigen lassen. Das funktioniert nur wenn unser Knoten nicht das Ende der Liste ist. Ist das der Fall so setzen wir den Knoten einfach zu NULL.

(2) Sei erneut eine einzeln verlinkte Liste gegeben. Um jetzt einen Pointer zu erstellen der beispielsweise genau auf die Mitte zeigt können wir folgendermaßen vorgehen:

Wir erstellen 2 Pointer. Der eine läuft immer ganz normal einen Schritt nach vorne, während der Zweite immer 2 Schritte macht. Das Ganze kapseln wir in eine while Schleife, die läuft solange der schnell Pointer nicht am Ende ist. Erreicht der schnelle Pointer jetzt das Ende so ist der langsame Pointer in der Mitte. Das geht deutlich schneller und effektiver als die naive Methode erst mit einem Pointer die Länge bestimmen und dann nochmal von vorne bis zur Mitte laufen.

(3) Statt eine extra Stack klasse zu implementieren benutzen wir einfach Vektoren und die dafür definierten Methoden $push\_back()$, $empty()$, pop_back und $back()$.

(4) Um eine Liste als queue zu verwenden benutzen wir die Methoden $emplace\_back(.)$, die ein Element hinten an die Liste anhängt. Zusätzlich benutzen wir $front()$ um das erste Element zu erhalten sowie $pop\_front()$ um das erste Element zu entfernen.

(5) Ein BinaryTree ist einfach ein Baum, indem jeder Knoten maximal 2 Kinder hat. Ein BinarySearchTree hingegen ist ein BinaryTree, bei dem für jeden Knoten gilt: Das Maximum des linken Teilbaumes ist kleiner oder gleich als die Wurzel des Teilbaumes. Das Minimum des rechten Teilbaumes ist größer als die Wurzel des Teilbaumes.

(6) Es gibt 3 verschiedene Möglichkeiten einen BinaryTree zu traversieren:

- Preorder: Hier betrachten wir zuerst den jeweiligen Knoten wo wir sind, gehen dann erst nach Links und machen hier wieder das Selbe und anschließend noch nach rechts. Da wir erst die Knoten immer betrachten kann man mit diesem Algorithmus gut eine Kopie des Baumes machen.
- Inorder: Wir gehen erst nach Links, wenn im linken Teilbaum alles abgearbeitet ist betrachten wir unseren Knoten und gehen anschließend noch in den rechten Teilbaum und wiederholen das Ganze. Da in einem BinarySearchTree links immer die kleinsten Werte stehen gibt uns dieser Algorithmus hierfür eine sortierte Liste mit allen Werten.
- Postorder: Bei postorder gehen wir erst von einem Knoten erst in den linken Teilbaum, dann in den Rechten und schließlich betrachten wir die Wurzel des

Teilbaumes. Diese Reihenfolge hat zur Folge, dass wir erst alles Kinder eines Knotens betrachten, bevor der Knoten selbst an der Reihe ist. Somit eignet sich dieser Algorithmus zum Löschen eines Teilbaumes.

(7) Ein trie ist im Prinzip ein Baum wo jeder Knoten beliebig viele Kinder haben kann. Die Wurzel ist natürlich ausgezeichnet. In jedem Knoten steht ein Charakter und somit repräsentiert jeder Pfad von der Wurzel zum Blatt ein Wort.

Ein radix Tree ist ähnlich aufgebaut, nur sind an den Knoten statt einzelner Buchstaben, Strings gegeben (Können auch einzelne Buchstaben sein). Somit haben wir einfach nur einen Speicher effizienteren trie.