Tutorial für Präfix-Summe

In diesem Tutorial wird ein Problem und der dazugehörige Lösungsweg beschrieben. Das Problem scheint sehr einfach zu sein, denn dieses kann sicherlich von sehr vielen Programmierern zumindest ineffizient gelöst werden. Eine ineffiziente Lösung ist jedoch nur für wenige Input-Daten geeignet. Viel spannender wird es, wenn der Input sehr groß wird, z.B. eine Million Zahlen. Dieses Tutorial beschäftigt sich genau damit, wie man die Lösung für dieses Problem auf effiziente Weise ermittelt, damit sie dann bei vielen Input-Daten auch noch schnell genug berechnet werden kann.

Nachdem fast alle bei der Österreichischen Informatik Olympiade in C++ programmieren, ist dieses Tutorial sehr stark an diese Sprache angelehnt.

Problembeschreibung

Angenommen, du hast eine Sequenz von Zahlen gegeben (in der Softwareentwicklung könnte man dies mit einem Array vergleichen, welches mit Zahlen gefüllt ist). Du möchtest nun die Summe verschiedener Bereiche dieser Sequenz ermitteln. Das Ermitteln der Summe eines Bereiches wird später als Abfrage bezeichnet. Folgendes Beispiel soll dies verdeutlichen:

Beispiel

Beispielsweise könnten eine Sequenz und die Abfragen nach der Summe wie folgt aussehen:

Sequenz:

Index:	0	1	2	3	4	5	6	7	8
Wert:	3	4	6	2	6	7	9	1	6

Abfragen:

- Abfrage von Index 0 bis Index 2: Summe = 3 + 4 + 6 = 13
- Abfrage von Index 0 bis Index 8: Summe = 3 + 4 + 6+ 2 + 6 + 7 + 9 + 1 + 6 = 44
- Abfrage von Index 2 bis Index 5: Summe = 6 + 2 + 6 + 7 = 21

Implementierungsdetails

Bei dieser Aufgabe werden die Werte von der Standardeingabe (Tastatur) gelesen und die Ergebnisse (Summen) auf die Standardausgabe (Bildschirm) geschrieben. Für dieses Problem sind die Ein- und Ausgabe jedoch schon implementiert. Es müssen nur mehr 2 Funktionen implementiert werden: `void init(int count, int sequence[])` und `long long getSum(int from, int to)`. Die Funktion `init` wird nur einmal am Anfang aufgerufen. Hier kann man sich die Werte kopieren und Initialisierungen vornehmen. Die Funktion `getSum` wird für jede Abfrage aufgerufen. Diese soll die Summe der Zahlen zwischen den Indizes `from` und `to` (jeweils inklusive) zurückgeben. Als C++ Datentyp für den Rückgabewert wird `long long` verwendet, da die Summe ziemlich groß werden kann.

Basislösung

Eine Implementierung, die das Problem löst und überhaupt keine Optimierung enthält nennt sich Basislösung und ist im Allgemeinen eher langsam.

Wie bereits erwähnt, scheint dieses Problem sehr einfach zu sein. Das liegt daran, dass die Basislösung sehr offensichtlich und leicht zu implementieren ist: Bei der Basislösung würde man für jede Abfrage die Zahlen in dem gefragten Intervall durchlaufen und addieren.

Der Code für die Basislösung würde wie folgt aussehen.

```
1    #define N_MAX 1000000
2
3    int N;
4    int A[N_MAX];
5
6    void init(int count, int sequence[]) {
7        N = count;
8        for (int i = 0; i < count; i++) {
9            A[i] = sequence[i];
10       }
11   }
12
13   long long getSum(int from, int to) {
14       long long sum = 0;
15
16       for (int i = from; i <= to; i++) {
17           sum = sum + A[i];
18       }
19
20       return sum;
21   }
```

Berechnung der Laufzeit[1]

Es wurde erwähnt, dass dieser Code nur eine Basislösung umsetzt und es schnellere Lösungsverfahren gibt. Jetzt stellt sich die Frage, wie stellt man fest, ob dieser Algorithmus schon schnell genug ist?

Zur Laufzeitabschätzung gibt es die Groß-O-Notation. Diese gibt Auskunft wie sich die Anzahl der Rechenschritte in Bezug auf die Anzahl der Eingabedaten verhalten. Z.B. $O(N^2)$ bedeutet, dass sich die Anzahl der Rechenschritte quadratisch zur Größe der Eingabedaten verhalten. D.h. im schlechtesten Fall benötigt man N^2 Rechenschritte, wenn der Input die Größe N besitzt. Anzumerken ist hier noch, dass konstante Faktoren weggelassen werden, d.h. $O(4*N^2)$ entspricht $O(N^2)$.

Generell kann man sagen, dass man ca. **10 Millionen Rechenschritte pro Sekunde** mit einem herkömmlichen Computer berechnen kann.

In folgender Tabelle sieht man einen Überblick über verschiedene Laufzeitkomplexitäten und deren Anzahl an Rechenschritten für unterschiedliche Eingabegrößen:

[1] In diesem und anderen Kapiteln wurden absichtlich mathematische Details weggelassen um die Erklärungen möglichst einfach zu halten.

	Anzahl der Rechenschritte bei jeweiliger Komplexität			
Eingabegröße N	**O(N)**	**O(N^2)**	**O(N^3)**	**O(2^N)**
10	**10**	**100**	**1 000**	**1 000**
100	**100**	**10 000**	**1 000 000**	10^{30}
1 000	**1 000**	**1 000 000**	10^9	10^{300}
10 000	**10 000**	100 000 000	10^{12}	unvorstellbar groß
100 000	**100 000**	10^{10}	10^{15}	
1 000 000	**1 000 000**	10^{12}	10^{18}	
10 000 000	**10 000 000**	10^{14}	10^{21}	
100 000 000	100 000 000	10^{16}	10^{24}	

Die fett markierten Einträge würden noch in einer Sekunde berechnet werden können.

Jetzt ist interessant, welche Laufzeit unsere Basislösung besitzt.

Laufzeit der Basislösung

Wie bereits erwähnt, berechnet man bei der Groß-O-Notation den schlechtesten Fall und lässt Details wie konstante Faktoren weg.

N bezeichnet die Anzahl der Werte der Sequenz, Q die Anzahl der Abfragen (Queries).

Bei unserem Algorithmus wird die `init`-Funktion genau einmal Aufgerufen. Diese enthält eine Schleife der Länge N. Die `getSum`-Funktion wird Q-mal aufgerufen und enthält wiederum eine Schleife, die im schlechtesten Fall N-mal durchlaufen wird. D.h. die Anzahl der Rechenschritte für unseren Algorithmus ist N + Q*N = (Q+1)*N. Nachdem wir konstante Faktoren weglassen, kann man sagen, dass die Basislösung eine Laufzeit von O(Q*N) besitzt.

Wenn man annimmt, dass es Q = N Abfragen geben kann, dann würde die Laufzeit O(N^2) sein. Aus obiger Tabelle kann man entnehmen, dass N nicht einmal 10 000 sein darf, damit dieses Programm innerhalb von 1 Sekunde alle Werte berechnet hat. Wenn N beispielsweise eine Million wäre, dann würde man 10^{12} Rechenschritte benötigen, um alle Abfragen zu berechnen. Bei 10^7 Rechenschritten pro Sekunde würde das Programm somit 10^5 = 100 000 Sekunden, also ca. 30 Stunden benötigen.

Somit sollte klar sein, dass es einen Unterschied macht, wie komplex der Algorithmus ist.

Effizienterer Algorithmus

Nun möchten wir einen Algorithmus vorstellen, der eine Laufzeit von O(N+Q) besitzt, also im Fall von Q = N eine lineare Laufzeit von O(N). Somit kann für N = 1 000 000 das Ergebnis in einer Sekunde berechnet werden und es werden nicht 30 Stunden benötigt, wie beim vorherigen Beispiel.

Der Trick ist, dass man sich bereits einige Werte vorberechnet, um dann die Abfrage in konstanter Zeit ermitteln zu können, also keine Schleife mehr benötigt.

Insgesamt gibt es ca. N^2 verschiedene Intervalle. Somit kann die Funktion `getSum` mit ca. N^2 verschiedenen Parameterwerten aufgerufen werden. Alle möglichen Intervalle im Vorhinein zu berechnen, ist keine gute Idee, da dies mindestens eine Laufzeit erfordert, die mit O(N^2) wächst, da es schon N^2 Intervalle gibt.

Viel besser ist die Idee mit der Präfix-Summe: Hier berechnet man sich die Summe aller Intervalle, die bei 0 beginnen. D.h. die Summe des Intervalls von 0 bis 0, die Summe des Intervalls von 0 bis 1, die Summe des Intervalls von 0 bis 2, usw.

In unserem Beispiel würde das wie folgt aussehen:

Index:	0	1	2	3	4	5	6	7	8
Wert:	3	4	6	2	6	7	9	1	6
Präfix-Summe:	3	7	13	15	21	28	37	38	44

Wenn man jetzt die Summe in einem Intervall berechnen möchte, muss man nur mehr 2 Werte subtrahieren. Wiederum unser Beispiel:

- Abfrage von Index 0 bis Index 2: Summe = 13 – 0 = 13
- Abfrage von Index 0 bis Index 8: Summe = 44 – 0 = 44
- Abfrage von Index 2 bis Index 5: Summe = 28 – 7 = 21

Mathematisch würde das wie folgt aussehen:

$$\sum_{i=from}^{to} A[i] = \sum_{i=0}^{to} A[i] - \sum_{i=0}^{from-1} A[i]$$

Die Summen auf der rechten Seite sind jeweils vorausberechnet, somit kann die Summe auf der linken Seite in konstanter Zeit ermittelt werden. Die `init`-Funktion benötigt die Laufzeit O(N) zum Kopieren der Werte und zum Vorberechnen der Präfix-Summe. Die `getSum`-Funktion wird Q-mal aufgerufen und benötigt jeweils die Laufzeit O(1). Gesamt ergibt sich die Laufzeit O(N + Q).

Der Code für die effiziente Lösung würde wie folgt aussehen:

```
1    #define N_MAX 1000000
2
3    int N;
4    int A[N_MAX];
5    long long prefixSums[N_MAX];
6
7    void init(int count, int sequence[]) {
8        N = count;
9        for (int i = 0; i < count; i++) {
10           A[i] = sequence[i];
11           if (i == 0) {
12               prefixSums[i] = A[i];
13           }
14           else {
15               prefixSums[i] = A[i] + prefixSums[i-1];
16           }
17       }
18   }
19
20   long long getSum(int from, int to) {
21
22       long long sum = prefixSums[to];
23
24       if (from > 0) {
25           sum = sum - prefixSums[from - 1];
26       }
27
28       return sum;
29   }
```

Es ist ersichtlich, dass jetzt nicht nur die Werte sondern auch die Präfixsummen gespeichert werden (Zeile 4 und 5). Nachdem man jedoch die normalen Werte in der `getSum`-Funktion nie benötigt, müsste man sie nicht einmal speichern.

Weiters sieht man, dass dieser Algorithmus für ein maximales N von einer Million funktioniert, da das Array diese Größe besitzt.

Bei uns in der Informatik Olympiade geht es vor allem um Algorithmen und Datenstrukturen. Die Aufgabe ist immer, ein Programm zu schreiben, welches eine gewisse Menge an Input-Daten verarbeitet und dabei schnell genug ist (meist unter 1 Sekunde) und nur einen begrenzten Speicher benötigt. Eine Variable vom Typ `long long` braucht auf den meisten Maschinen 8 Byte Speicherplatz, eine Variable vom Typ `int` nur 4 Bytes. Somit benötigt unser Algorithmus ca. 12 Millionen Bytes Speicherplatz, also ca. 12 Megabytes. Nachdem wir bei der Problemstellung diesen Speicher nicht festgelegt haben, gehen wir jetzt einmal davon aus, dass dieser zur Verfügung steht.

Theoretisch könnte man das Array auch dynamisch allokieren und genau so groß machen, wie es wirklich nötig ist. Nachdem es bei der Informatik Olympiade eine allgemeine Speicherobergrenze gibt, die eingehalten werden muss, und es egal ist, ob diese bei kleinen Eingabedaten auch schon fast ausgeschöpft ist, können wir ruhig alles statisch allokieren.

Tipp: Verwende globale Arrays, denn diese sind in C++ automatisch mit 0 initialisiert.

Kurz gesagt: Es geht nicht zwingend um „Schönheit“ bei diesem Wettbewerb. Klar, sollte der Algorithmus einigermaßen lesbar sein, damit man sich selbst noch auskennt, aber ob z.B. eine Variable global oder lokal definiert ist, ist zumindest bei diesem Wettbewerb egal.

Schlussworte

Wir, die Betreuer der Österreichischen Informatik Olympiade, hoffen Dich mit diesem Tutorial für die Effizienz von Algorithmen begeistert zu haben und würden uns freuen, wenn Du versuchst die Beispiele für die Qualifikation zu lösen.

Tipp: Möglicherweise kann dir dieses Tutorial dabei helfen, die Qualifikationsaufgaben möglichst gut zu lösen.

Vielleicht bist Du dann beim ersten Trainingscamp dabei und lernst viele weitere spannende Algorithmen kennen.

Viel Spaß dabei!