Dokumentation der Implementierung der Funktion MERGE

Bearbeitungszeit:

Recherche und Implementierung des Algorithmus	ca. 1 Stunde
Test & Verbesserungen am Algorithmus	ca. 1,5 Stunden
Aufsetzen der Entwicklungsumgebung, Repo, Readme etc.	ca. 0,5 Stunden
Dokumentation und Ermittlung der relevanten Daten	ca. 2 Stunden
Gesamt	**ca. 5 Stunden**

1. Allgemeine Annahmen

Folgende Annahmen habe ich als Entwickler getroffen:

- Die übergebenen Intervalle sind nicht vorsortiert, sondern können in einer beliebigen Reihenfolge übergeben werden.
- Die Funktion MERGE erhält die Intervalle als Liste.
- Jedes Interval [x,y] ist ein Int32 Array mit zwei Elementen x und y, wobei das Programm auch prüfen muss, ob die einzelnen Intervalle valide sind und falls nicht, muss es entsprechend damit umgehen.
- x ist immer kleiner als y. Sollte das nicht der Fall sein, ist das Interval nicht valide und das Programm muss damit umgehen können.

2. Entscheidungen über das Design der Anwendung

Folgende Entscheidungen habe ich als Entwickler getroffen:

- Um das Bauen und Ausführen der Anwendung so einfach wie möglich zu gestalten, habe ich mich dazu entschieden, die Anwendung in C# zu programmieren. Nach meiner Erfahrung ist das Aufsetzen einer Entwicklungsumgebung hier am einfachsten. Alles was man benötigt, um die Applikation auszuführen, ist eine Installation des .NetCore v3.1 Frameworks und Visual Studios. Näheres dazu in der README oder im nachfolgenden Kapitel.
- Auf ein Übergeben der Intervalle über ein grafische Benutzeroberfläche oder das Einlesen aus einer Datei oder über die Argumente in der Main-Methode des Programms, habe ich aus Zeitgründen verzichtet. Stattdessen werden die zu verarbeitenden Intervalle in der Main-Methode der Hauptanwendung oder in den Unit-Tests definiert.

3. Bauen und Ausführen der Anwendung

Nachfolgend sind die Schritte aufgelistet, die durchgeführt werden müssen, um das Programm zu öffnen, zu bauen und auszuführen:

- Benötigt wird eine Installation des .NetCore3.1 Frameworks. Diese steht hier zum Download bereit. Meistens ist diese Abhängigkeit bereits installiert. Sollte das Programm beim Ausführen aber eine entsprechende Fehlermeldung ausgeben, fehlt auf dem Zielsystem .NetCore 3.1.
- Eine Installation der IDE Visual Studio. Diese ist hier zu finden.
- Das Versionsverwaltungswerkzeug Git muss auf dem Zielsystem installiert sein. Dieses kann hier heruntergeladen werden, falls nicht vorhanden.

Sind alle Abhängigkeiten installiert, kann mithilfe von Visual Studio und Git das Repository geklont werden:

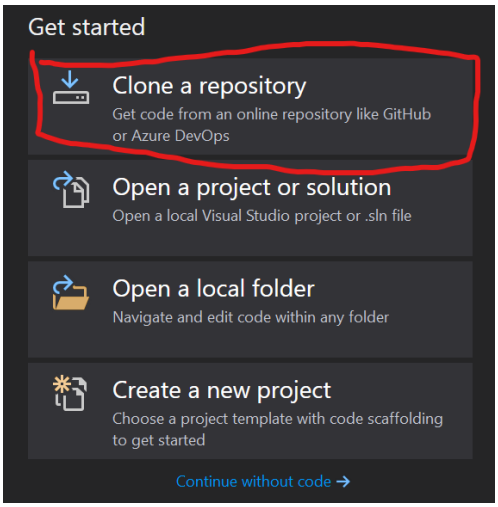

Abbildung 1: Klonen eines Repositories

Dabei öffnet sich sein neues Fenster, in welches die URL des Repositories sowie ein Zielordner eingefügt werden muss:

https://github.com/chgsit00/interval-merge.git

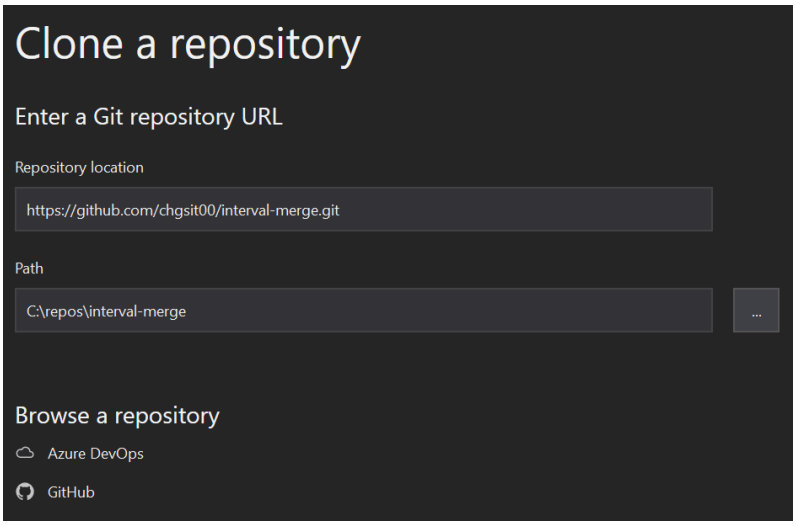

Abbildung 2: Einfügen des Links zum Repository

Nach Bestätigung der Eingaben, lädt Visual Studio die Solution, sowie die zugehörigen Projekte. Mithilfe des Run-Buttons in der oberen Leiste Visual Studio kann das Programm gebaut und ausgeführt werden:

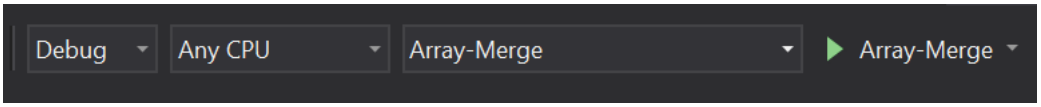

Abbildung 3: Run-Button

4. Aufbau des Algorithmus

Bei der Implementierung des Algorithmus der Merge-Funktion waren mir folgende Punkte wichtig:

- Um die Komplexität des Codes niedrig zu halten und eine unnötige Beanspruchung des CPU zu vermeiden, sollten mehre verschachtelte for- oder while-Schleifen vermieden werden. Stattdessen sollte der Algorithmus nur ein einziges Mal über die Liste der übergeben Intervalle iterieren müssen.
- Um den Speicherbedarf niedrig zu halten, sollte der Algorithmus mit so wenig Kopien der Intervall-Liste wie möglich auskommen. Da diese beliebig lang sein kann, würden mehrere Kopien den RAM-Verbrauch in die Höhe treiben.
- Damit der Algorithmus nicht jedes Element der Liste mit jedem anderen Element der Liste vergleichen muss, sollte die Liste vorsortiert werden, sodass immer nur ein Listenelement mit dem vorherigen abgeglichen werden muss.

Daraus folgte folgender Algorithmus, wobei der Algorithmus annimmt, dass jedes Intervall in der übergebenen Liste ein Int32 Array mit zwei Elementen [x, y] ist:

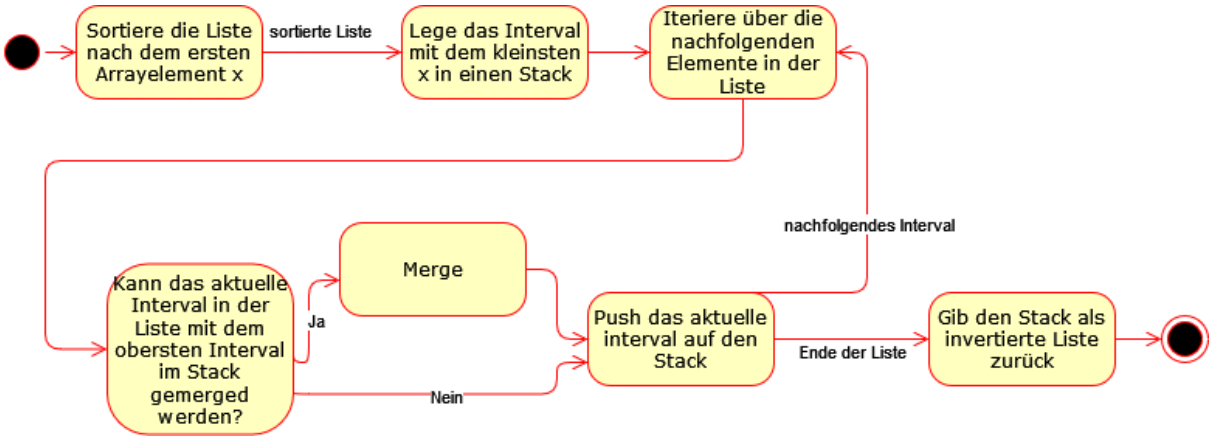

Abbildung 4: Aktivitätsdiagram des Algorithmus

Die Entscheidung, ob zwei Intervalle gemerged werden können, wird anhand der Überlappung des Endes des ersten Intervalls ***A*** mit dem Anfang des zweiten Intervalls ***B***, getroffen. Ist ***B[x]*** kleiner oder gleich ***A[y]***, muss ein Merge durchgeführt werden.

Bei einem Merge wird ein neues Array ***C*** erstellt, wobei der erste Element ***C[x]*** des neuen Intervalls den Wert des Anfangselements **x** des ersten Intervalls ***A[x]*** erhält. Das zweite Element ***C[y]*** dagegen erhält den Wert des zweiten Elements des zweiten Intervalls ***B[y]***. Daraus folgt das neue Intervall:

$$C[x, y] = C[A[x], B[y]]$$

Das neue Intervall ***C*** wird nun auf den Stack gepusht und nimmt den Platz des alten Intervalls ***A*** ein, um mit dem Nachfolgenden Intervall verglichen zu werden.

Weitere Anmerkungen:

- Sind die Bedingungen für einen Merge nicht erfüllt, dann wird das aktuelle Intervall ***B*** auf den Stack gepusht. Dadurch wächst der Stack durch das Hinzufügen von neuen Intervallen.
- Da der Stack absteigend ausgelesen wird, muss er am Ende des Algorithmus invertiert werden, um die Reihenfolge vom Intervall mit dem kleinsten x bis zum Intervall mit dem größten x wiederherzustellen.

5. Verbesserung des Algorithmus und Tests

Um das Programm stabiler und robuster zu machen, wurde der Algorithmus auf die nachfolgenden Kriterien geprüft.

5.1 Robustheit

Unter der Robustheit versteht man die Fähigkeit eines Programms, bei fehlerhafter Bedienung oder bei fehlerhaften Eingaben einen stabilen Zustand beizubehalten. Um die Robustheit eines Programms zu überprüfen, werden für gewöhnlich Unittests geschrieben, die mit Absicht falsche Werte oder Werte in einem extremen Spektrum an das Programm übergeben.

Um dies zu tun, muss man sich ansehen, welche falschen Eingaben möglich sind, die die Stabilität des Programms beeinträchtigen könnten. Folgende Eingaben wurden dabei erkannt:

1. Die Liste der Intervalle wird unsortiert übergeben. Z.B. [3,6] [1,2] [13,20] [8, 14]
   Lösung: Die Liste wird vor dem Durchlaufen des Algorithmus sortiert. Siehe Kapitel 4

2. Es wird nur ein oder gar kein Intervall mit der Liste übergeben.
   Lösung: Die Anzahl der Elemente in der Liste wird vorher geprüft und das Programm gibt eine Warnung aus, sollte die Anzahl der Intervalle unter 2 sein.

3. Eines oder mehrere der Intervalle hat die falsche Anzahl an Elementen. Z.B. [0] oder [1,3,5]
   Lösung: Der Algorithmus untersucht immer nur das erste und das letzte Element innerhalb eines Intervalls. Dazwischen liegende Elemente werden ignoriert. Z.B. aus [1,3,5] wird [1,5].
   Alternative: Der Algorithmus bricht ab und gibt eine Fehlermeldung aus.
   Alternative 2: Der Algorithmus untersucht immer die niedrigste und die höchste Zahl in einem Intervall. Z.B. aus [3, 1, 9, 4] wird [1, 9]. Dies würde allerdings mehr Rechenaufwand bedeuten.

4. Eines der Intervalle A hat das falsche Format, sodass A[x] größer ist als A[y]. Z.B. [9,4]
   Lösung: Der Algorithmus bricht ab und gibt eine Fehlermeldung aus.
   Alternative: Siehe 3. Alternative 2.

5. Es werden extrem hohe oder niedrige Werte innerhalb eines Intervalls übergeben, wie Z.B. int.MaxValue oder int.MinValue.
   Lösung: Die Sortieren-Funktion des Algorithmus muss entsprechend angepasst werden, um mit derart extremen Werten umgehen zu können.

6. Es werden extrem viele Intervalle übergeben.
   Lösung: Stresstest mit einer Funktion, die autogenerierte Intervalle erzeugt, dabei Messung der Performance.
   Alternative: Obergrenze für Anzahl der Intervalle festlegen und wenn diese überschritten wird, eine Fehlermeldung ausgeben.

7. Es werden Arrays des falschen typs übergeben. Z.B. Long statt Int32.
   Lösung: Typsicherheit gewährleisten. (In diesem Fall nicht nötig, da der Methodenkopf in C# bereits auf den Typ der Arrays prüft)

5.1 Laufzeit

Die Laufzeit des Programms in erster Linie von der Komplexität des Algorithmus, sowie von der Anzahl der übergebenen Intervalle ab. Dies lässt sich recht einfach mit der O-Notation für die Bestimmung der Algorithmen bestimmen. Im Optimalfall wird nur ein einziges Mal über die Intervalle iteriert:

$$O(N)$$

Wobei N die Anzahl der Intervalle in der Liste darstellt. Je öfter durch die Listenelemente iteriert wird, desto größer die Komplexität. Wird z.B. a mal über die Liste iteriert, ändert sich die Komplexität wiefolgt:

$$O(a * N)$$

Kommen noch verschachtelte Schleifen ins Spiel, die innerhalb einer Schleife noch einmal über die Liste iteriert, wird die Komplexität sogar quadriert:

$$O(N^2)$$

Daher gilt es, hier eine Verschachtelung zu vermeiden. Da der Algorithmus nur zweimal über die Liste iteriert, einmal, um die Liste vorher zu sortieren und einmal, um das Mergen durchzuführen, entspricht die Komplexität in etwa:

$$O(2N)$$

Da die Laufzeit stark an die Komplexität gekoppelt ist, ist zu erwarten, dass die Laufzeit mit der Anzahl der Listenelemente linear zunimmt. Dies ist auch der Fall:

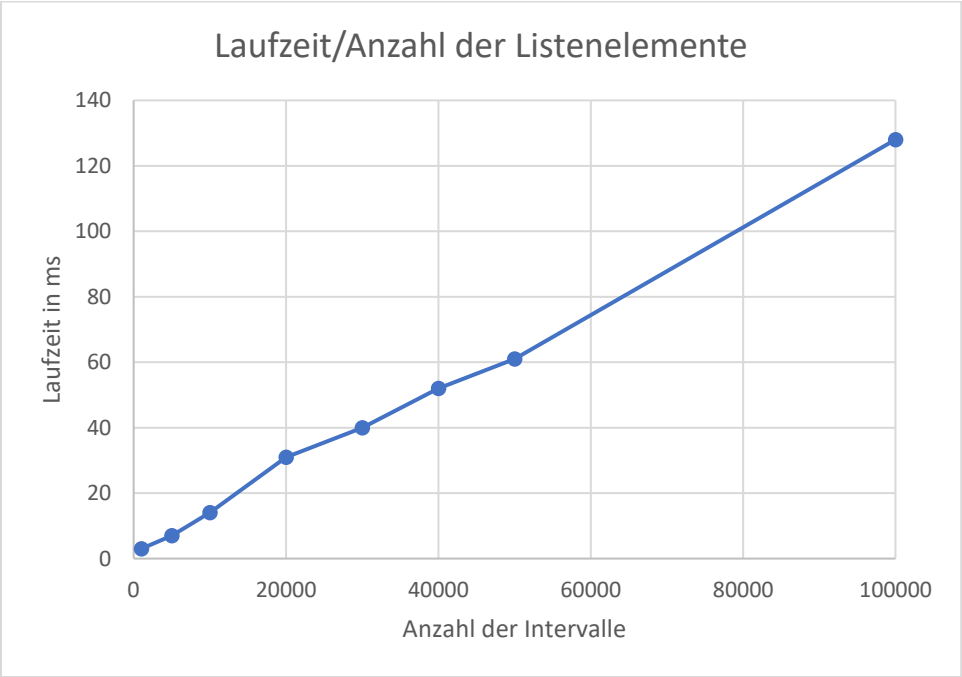

Abbildung 5: Abhängigkeit der Laufzeit von der Anzahl der Listenelemente

5.2 Speicherverbrauch

Der Speicherverbrauch der Anwendung lässt sich anhand der der lokal definierten Variablen bestimmen. Maximal sollte der Speicherverbrauch kurz vor Ende des Algorithmus am höchsten sein, da sowohl die übergebene Liste mit den Intervallen, als auch der Stack mit den zusammengefügten Intervallen gleichzeitig im RAM gehalten werden muss. Da beide stark von der Anzahl der Elemente in der Liste abhängig sind, wird erwartet, dass der Speicherverbrauch anfangs in etwa der Größe der Liste in Bytes entspricht und über den Verlauf des Algorithmus linear auf annähernd das Doppelte ansteigt, da nach und nach der Stack befüllt werden muss. Um dies zu prüfen, wurde am Ende der Verarbeitung ein Breakpoint gesetzt und die Applikation mit einen Liste von 1000000 Intervallen gestartet. Das Ergebnis des Visual Studio Diagnose-Tools ist im nachfolgenden Bild zu sehen:

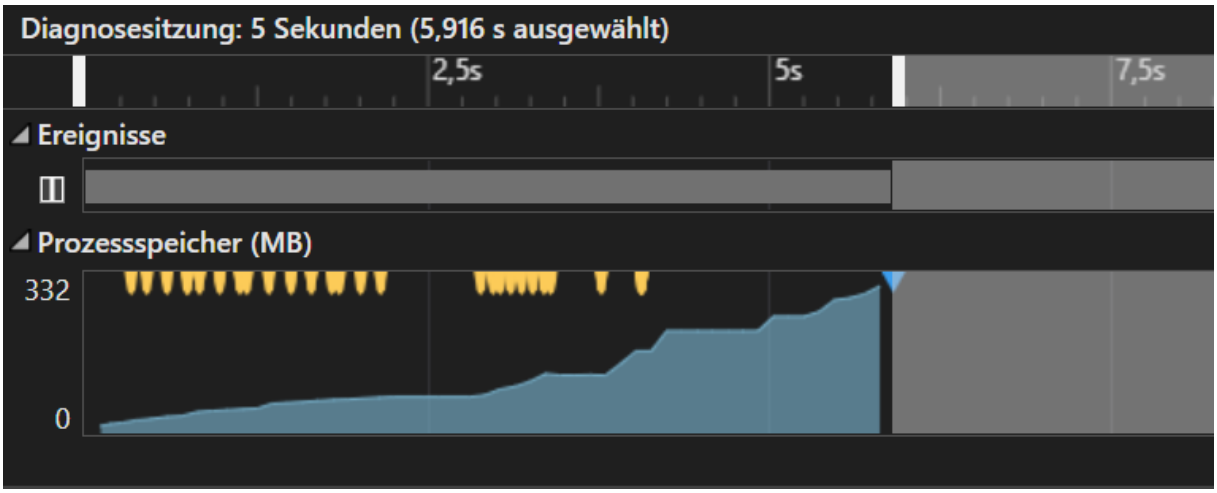

Abbildung 6: Speicherverbrauch der Anwendung zur Laufzeit

Wie man sieht, wächst der Speicherbedarf kontinuierlich an, bis das Maximum beim Breakpoint erreicht wird.