LEHRSTUHL FÜR RECHNERTECHNIK UND RECHNERORGANISATION

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Projektaufgabe - 323: Berechnung von Primzahlen

Marcel Zurawka
Julius Krüger
Lars Hinnerk Grevsmühl

1 Einleitung

Ziel des Praktikums zu Aspekte der systemnahen Programmierung bei der Spieleentwicklung ist die Bearbeitung einer Aufgabe aus dem Bereich der Algorithmik. Im folgenden betrachten wir unsere Assembler Implementierung der Wormell'schen Formel unter den Aspekten der Herleitung, Nutzung und Performanz und geben im Anschluss einen Ausblick auf alternative Lösungen und weitere Optimierungsvorschläge.

2 Problemstellung und Spezifikation

Ziel des Praktikums ist die Erstellung einer gut dokumentierten und optimierten Implementierung der Wormell'schen Formel (vgl. Formel 1) in Assembler. Diese kann allein unter Verwendung von Ganzzahlen zur Berechnung der n-ten Primzahl genutzt werden.

$$p_n = \frac{3}{2} + 2^{n-1} - \frac{1}{2}\sum_{m=2}^{2^n} (-1)^{2^{\prod_{r=1}^{n}\left(1-r+\frac{m-1}{2}+\frac{1}{2}\sum_{x=2}^{m}(-1)^{2}\prod_{a=2}^{x}\prod_{b=2}^{x}(x-ab)^{2}\right)^{2}}} \tag{1}$$

Bevor wir zum genauen Ziel unserer Arbeit kommen, ist darauf hinzuweisen, dass die durch das Aufgabenblatt gestellte Formel fehlerhaft ist. Auf dem Blatt ist sie notiert als "$p_n = 2 + 3/2 + ...$". Aus der Literatur [1] lässt sich allerdings entnehmen, dass die Wormell'sche Formel korrekt "$p_n = 3/2 + ...$"lautet, also ohne den ersten Summanden. Wir vermuten, dass dieser aus der Vermischung zweier Schreibweisen entstanden ist. Betrachtet man die Teilformeln ohne diese zusammenzufassen, so zeigt sich tatsächlich eine 2 als erster Summand (vlg. Formel 10). Im Folgenden arbeiten wir mit der allgemein üblichen Formel, nicht zuletzt da diese korrekte Ergebnisse liefert, wie unsere Tests ergeben haben.

Hierbei legen wir weniger Wert auf eine unveränderte Darstellung der Wormell'schen Formel, sondern viel mehr auf eine optimierte Abwandlung dieser, um eine realistische Anwendungsmöglichkeit zur Berechnung von kleineren Primzahlen zu bieten. Zur logischen Optimierung lassen wir uns daher die Freiheit Redundanzen zu reduzieren und irrelevante Tests mit besseren Ober- und Untergrenzen für unsere Schleifen

zu verhindern. Des Weiteren soll die technische Umsetzung von der Möglichkeit der Parallelisierbarkeit der Formel profitieren.

3 Lösungsfindung

3.1 Analyse der Wormell'schen Formel

Die hier genutzte Darstellung (vgl. Formel 1) weist nur wenig auf den intuitiv dahinter befindlichen Sachverhalt hin. Daher haben wir sie im folgenden in kleinere Teilformeln zerlegt. Setzt man diese nacheinander in $p(n)$ ein, erhält man dabei die dargestellte Form. Im Folgenden sollen diese Formeln rückwärts, also vom innersten hin zum äußersten Term, erläutert werden.

$$f(x) = \prod_{a=2}^{x} \prod_{b=2}^{x} (x - ab)^2 \tag{2}$$

$$f(x) = \begin{cases} 0 & \text{wenn } n \text{ nicht prim} \\ N \in \mathbb{N} & \text{wenn } n \text{ prim} \end{cases} \tag{3}$$

Das innerste Produkt (siehe Formel 2) lässt die zwei Variablen a und b von 2 bis n durchlaufen und prüft, ob eines der Produkte unserer gesuchten Zahl x entspricht. Denn auch wenn das Produkt beliebige Werte liefern kann, ist lediglich von Interesse, ob dieses gleich Null ist. Wenn dies der Fall ist haben wir zwei Zahlen gefunden deren Produkt x ergibt. Es kann sich also nicht um eine Primzahl handeln. Bei Primzahlen ergibt das Produkt einen positiven Ganzzahlwert.

$$g(m) = \sum_{x=2}^{m} \frac{1 + (-1)^{2^{f(x)}}}{2} = \frac{m-1}{2} + \frac{1}{2} \sum_{x=2}^{m} (-1)^{2^{f(x)}} \tag{4}$$

$$g(m) = \sum_{x=2}^{m} \begin{cases} 0 & \text{wenn } f(x) = 0 \\ 1 & \text{wenn } f(x) \in \mathbb{N} \end{cases} \tag{5}$$

Die Funktion $g(m)$ (siehe Formel 4) zählt die oben genannten Spezialfälle. Für jede Zahl von 2 bis m wertet sie die oben genannte Funktion aus. Handelt es sich um eine Null (falls x nicht prim), so wird auf die Summe auch eine Null aufaddiert. Handelt es sich um eine positive Ganzzahl (falls x prim), so addiert sie eine Eins auf. Sie zählt also die Primzahlen von 2 bis m.

$$h(m, n) = \prod_{r=1}^{n} (1 - r + g(m))^2 \tag{6}$$

$$h(m, n) = \begin{cases} 0 & \text{wenn } n > g(m) \\ N \in \mathbb{N} & \text{wenn } n \leq g(m) \end{cases} \tag{7}$$

Das Produkt $h(m,n)$ (siehe Formel 6) vergleicht zwei Werte. Wieder spielt dabei der Fakt eine Rolle, ob einer der Faktoren Null wird. Da es sich sowohl bei r, als auch bei $g(m)$ um positive Ganzzahlen handelt, wird einer der Faktoren Null, nämlich jener, bei dem r genau einen kleiner ist als $g(m)$. Dies tritt lediglich dann nicht auf, wenn $n \leq g(m)$ ist. Ist also n größer als $g(m)$, so ist der Rückgabewert 0, andernfalls handelt es sich beim Rückgabewert um eine positive Ganzzahl.

$$i(n) = \sum_{m=2}^{2^n} \frac{1 + (-1)^{2^{h(m,n)}}}{2} = \frac{2^n - 1}{2} + \frac{1}{2} \sum_{m=2}^{2^n} (-1)^{2^{h(m,n)}} = -\frac{1}{2} + 2^{n-1} \frac{1}{2} \sum_{m=2}^{2^n} (-1)^{2^{h(m,n)}} \tag{8}$$

$$i(n) = \sum_{m=2}^{2^n} \begin{cases} 0 & \text{wenn } h(m,n) = 0 \\ 1 & \text{wenn } h(m,n) \in \mathbb{N} \end{cases} \tag{9}$$

Bisher ermitteln wir also, ob unter einer Zahl bereits ausreichend Primzahlen liegen. Ist dies noch nicht der Fall erhalten wir eine positive Ganzzahl. Sonst eine Null. Wieder verwenden wir den Trick, den wir auch bei $g(m)$ angewendet haben und zählen alle Fälle in denen wir positive Ganzzahlen haben für den Bereich von 2 bis 2^n. Wir enden bei 2^n, da die n-te Primzahl nach Bertrand's Postulat[2] ($p_{n+1} < 2 * p_n$) und der ersten Primzahl ($p_1 = 2^1$) zwischen 2 und 2^n liegen muss. Wir zählen also alle Zahlen, die kleiner als p_n sind, wobei wir die 1 und p_n selbst ausschließen.

$$p(n) = 2 + i(n) = p_n \tag{10}$$

Um nun p_n zu erhalten addieren wir einfach die zwei Fälle 1 und p_n selbst hinzu und erhalten dann p_n, da jede natürliche Zahl trivialerweise gleich der Anzahl aller darunter liegenden, sich selbst eingeschlossen ist.

3.2 Abwandlung der mathematischen Darstellung

Zur Erläuterung der Implementierung wollen wir die Wormell'sche Formel im Folgenden in zwei aufteilen. Für den weiteren Verlauf beschreibt die innere Formel $f(x)$ und die äußere die Formeln $p(n)$ bis $g(m)$. Bei der Herleitung der Wormell'schen Formel scheint es nie von Relevanz gewesen zu sein Fälle die mehrfach getestet werden nur einmal zu testen und Fälle die sich leicht ausschließen lassen zu ignorieren. Daher beschäftigen sich die folgenden beiden Abschnitte vor allem mit der Reduzierung von Redundanz und der Suche nach passenden Ober- und Untergrenzen für die Berechnungen.

3.2.1 Äußere Formel

Der Sinn der äußeren Formel ist es zu zählen, wie oft die innere Formel eine Primzahl gefunden hat und zu prüfen, ob diese Anzahl bereits so groß ist wie unser gesuchtes n, da in diesem Fall alle weiteren gefundenen Primzahlen bis n^2 ignoriert werden müssen. Aufgrund der mathematisch geschlossenen Form wird die innere Formel allerdings ($2^n -$

$1) \cdot (n/2(n+1))$ mal aufgerufen. Die Anzahl dieser Aufrufe lässt sich stark vermindern, in dem wir beginnend mit der 2 für jede Zahl prüfen, ob es sich um eine Primzahl handelt und entsprechend einen Zähler erhöhen. Diese Vorhergehensweise entfernt alle redundanten Aufrufe der inneren Formel.

```
int outer(unsigned int n){
    unsigned int count = 0;
    unsigned int i = 2;
    while(1){
        if(isPrime(i)){
            counter++;
            if(count==n){
                break;
            }
        }
        i++;
    }
    return i;
}
```

3.2.2 Innere Formel

Zuerst ist zu erwähnen, dass es völlig ausreicht eine einzige Ganzzahl-Faktorisierung zu finden. Danach kann es sich nicht um eine Primzahl handeln und wir können die Suche abbrechen. Des Weiteren ist es bei der Optimierung der inneren Formel vor allem wichtig geeignete Ober- und Untergrenzen für die Laufvariablen a und b zu finden.

Als Obergrenze für beide eignet sich $n/2$ sehr gut. Zum einen ist es die kleinstmögliche Obergrenze, da die kleinste Zerlegung $n = 2 \cdot n/2$ ist, und zum anderen lässt sie sich durch einen Shift nach rechts sehr schnell berechnen. Die größtmögliche Untergrenze für a bleibt 2, da wir nach wie vor wenigstens einen der beiden Faktoren der Zerlegung durchlaufen lassen müssen. Die größtmögliche Untergrenze für b ist n/a. An dieser Stelle bieten sich uns drei Möglichkeiten. Entweder wir sparen uns den Aufwand eine Untergrenze zu finden und fangen weiter bei 2 an, wir nähern uns a/n mit einer möglichst schnellen Methode an (mehr dazu unter 3.3) oder wir berechnen a/n exakt. Letzteres ist sehr rechenaufwändig, erspart uns allerdings auch den vollständigen Schleifendurchlauf von b, da wir einfach prüfen können ob a/n eine Ganzzahl ist, da dann unser Ergebnis zusammen mit a eine Faktorisierung von n ist.

```
unsigned int f(unsigned int x){
    unsigned int xHalf = x/2;
    float xa;
    for(unsigned int a=2; a<=xHalf; a++){
        xa = x/a;
        if(isInteger(xa)){
            return 0;
        }
```

```
9       }
10      return 1;
11  }
```

3.3 Annäherung der Division

An dieser Stelle wollen wir kurz darauf eingehen, warum wir uns gegen eine Annäherung der Division entschieden haben. Wenn man die Division von unten annähert und damit einen geeigneteren Startwert als 2 für b findet, so ist zu erwarten, dass je nach Präzision der Annäherung b noch etliche Schleifendurchläufe durchgeführt werden müssen. Als Test sei dazu folgende Annäherung gegeben:

```
1  unsigned int apprxDivide(unsigned int x, unsigned int a){
2      unsigned int b = 1;
3      while(a<=x){
4          x /= 2;
5          b *= 2;
6      }
7      return b;
8  }
```

Man beachte dabei, dass die Division bzw. Multiplikation um zwei jeweils durch einfache Shifts umzusetzen sind. Testen wir jedoch diese Version für $n = 1 \cdot 10^4$, so benötigt sie rund 43 Sekunden, wohingegen die exakte Division lediglich rund 4,5 Sekunden benötigt. Aufgrund des drastischen Leistungsunterschiedes haben wir diese Möglichkeit verworfen. Möglicherweise erhält man deutlich bessere Ergebnisse bei einer feineren Annäherung der Division. Mehr dazu unter 6.

4 Dokumentation der Implementierung

4.1 Benutzer-Dokumentation

Falls die finale Version des Programms genutzt werden soll, findet man dieses unter PrimeCalculationAssembler. Alternativ existieren unter Tests Versionen zum Vergleich zum C basierten Programm bzw. unter OlderVersions andere Versionen des Programms, die hauptsächlich Variationen in der Wahl der Grenzen, der Schleifen und der inneren Formel aufzeigen. Im jeweiligen Ordner ist die Variation des Programms nthPrime aus zu führen.

Zur Anwendung des Programms führt der Nutzer dieses mit dem notwendigen Parameter n aus. Hierbei sollte n eine Ganzzahl sein die größer als 0 ist. Für die maximale Größe von n ist zu beachten, dass p_n nicht größer als 2^{32} werden sollte. Als Richtwert wäre hier nach Bertrand's Postulat[2] die Zahl 32 als Obergrenze zu wählen. Da diese

deutlich zu klein ist können wir über vorherige Berechnungen zeigen, dass n sogar über $2 \cdot 10^8$ groß werden kann. Je nach Rechenleistung können aber bereits deutlich kleinere Werte von n unpraktikabel werden. Genaueres dazu ist unter 5.1 zu lesen. Die Ausgabe des Programms liefert die n-te Primzahl, sowie die zur Berechnung benötigte Zeit.

4.2 Entwickler-Dokumentation

Bevor wir uns mit dem Ablauf des Assembler Programms selbst befassen, ist es sinnvoll aufzuzeigen welche unserer Register welcher Aufgabe zugeordnet sind. Hierbei ist zu beachten, dass wir unter anderem drei q-Register verwenden, um später die Division a/n vier mal parallel ausführen zu können. Im Folgenden werden wir die entsprechenden Register nicht bei ihrer Nummer, sondern ihrem Namen nennen, um das Verständnis zu erleichtern.

Register	Name	Funktion
r0	n	Nummer der gesuchten Primzahl p_n
r1	x	Laufvariable, vermutetes p_n
r2	$primeCounter$	Anzahl der bereits gefundenen Primzahlen
r3	a	vermuteter Teiler von x
r4	tmp_1	Zwischenspeicher zum Laden der q-Register
r5	$x/2$	Obergrenze für a
r6	$divisorCounter$	Anzahl der gefundenen Teiler
r7	tmp_2	Zwischenspeicher zum Laden der q-Register
q0	a_{1-4}	Divisor
q1	x_{1-4}	Divident
q2	$result_{1-4}$	Quotient aus x_{1-4} und a_{1-4}

Am Anfang des Programms initialisieren wir unsere Variablen und setzen einige Startwerte ($primeCounter = 0; x = 1$). Unser finaler Assembler Code enthält zwei Schleifen. Die äußere Schleife $trueloop$ stellt dabei unsere äußere Formel (siehe 3.2.1) dar. In ihr werden zu Beginn weitere Startwerte gesetzt ($x_{1-4} = x, x, x, x; a_{1-4} = 2, 3, 4, 5; a = 2; x/2 = \frac{x}{2}, divisorCounter = 0$). Danach wird die innere Schleife $aloop$ aufgerufen, die unsere innere Formel (siehe 3.2.2) darstellt. In ihr testen wir, ob unser a das aktuelle x teilt. Dabei teilen wir die Werte aus x_{1-4} durch die aus a_{1-4} und schreiben das Ergebnis in $result_{1-4}$. Anschließend betrachten wir diese Ergebnisse einzeln, casten sie zurück in Integer und prüfen, ob unser Ergebnis multipliziert mit dem dazugehörigen a aus a_{1-4} x ergibt. Ohne Parallelisierung kann man sich den Prozess vorstellen wie im folgenden Quelltext.

```
1 int isDivisor(unsigned int x, unsigned int a){
2     unsigned int b;
3     float tmp1 = (float) a;
4     float fx = (float) x;
5
6     tmp1 = fx/tmp1;
7     b = (int) tmp1;
8
```

```
9       if(b*a==x){
10          return true;
11      }
12      return false;
13  }
```

Nachdem wir nun alle Werte aus den q-Registern ausgewertet haben, zählen wir die Anzahl der erfolgreichen Faktorisierungen. Finden wir mindestens eine, kann es sich bei x nicht mehr um eine Primzahl handeln und wir kehren zu Beginn der äußeren Schleife $trueloop$ zurück, um das nächste x zu testen. Wurde keine gefunden, so kann natürlich noch ein höheres a eine Faktorisierung von x aufzeigen. Haben wir also noch nicht unser höchstes a bei $x/2$ erreicht, kehren wir zu der inneren Schleife $aloop$ zurück und erhöhen dieses. Falls wir $x/2$ überschritten haben, können wir uns sicher sein, dass x eine Primzahl ist. Wir erhöhen daher unseren $primeCounter$ um eins. Falls dieser nun unser gesuchtes n erreicht hat können wir bereits unser p_n zurückgeben. Andernfalls müssen wir erneut zu $trueloop$ zurückkehren und das nächste x prüfen.

5 Ergebnisse

Im Folgenden gehen wir näher auf die Testergebnisse unseres Programms ein. Wir haben dabei zum einen auf Korrektheit der berechneten Primzahlen getestet und zum anderen auf die Geschwindigkeit unseres Programms bei der Berechnung der Primzahlen.

Die Korrektheit unseres Programms haben wir durch den Vergleich mit den Ergebnissen anderer, bereits etablierter, Rechenprogramme [3] geprüft. Für die Werte mit denen wir dies getestet haben sind dabei keine Fehler aufgetreten und dementsprechend gehen wir davon aus, dass unser Programm die n-te Primzahl korrekt berechnen kann für alle Primzahlen < 2^{32}. Diesen speziellen Fall haben wir aufgrund der Dauer der Berechnung nicht getestet, wir sehen allerdings keinen Grund warum er nicht funktionieren sollte. Um die Geschwindigkeit zu ermitteln haben wir unser Programm mit einem C-Programm, welches in Optimierungsstufe O3 kompiliert wurde, getestet.

5.1 Rechendauer

Im Rahmen unserer Tests auf Geschwindigkeit haben wir uns bei k Berechnungen der n-ten Primzahl die schnellste, die langsamste und die durchschnittliche Rechendauer ausgeben lassen. Die hierbei auftretenden Schwankungen liegen zum größten Teil in einem Bereich unter 1%, weshalb wir diese im Weiteren vernachlässigen.

Die unten stehende Tabelle zeigt die Berechnung der n-ten Primzahl, die Versuchsgröße k, die durchschnittliche Zeit die jeweils in unserem Assembler-Code und dem O3 kompilierten C-Code für den Versuch benötigt wurde und zuletzt einen Faktor um den unser Code schneller ist, als der Compiler-Code.

n-te Primzahl	k	Zeit ASM (Avg.)	Zeit C (Avg.)	C/ASM
$1 * 10^2$	100	$5,270 * 10^{-4}$s	$6,250 * 10^{-4}$s	1,185
$1 * 10^3$	100	$3,432 * 10^{-2}$s	$4,158 * 10^{-2}$s	1,211
$1 * 10^4$	100	$4,520 * 10^{0}$s	$5,461 * 10^{0}$s	1.208
$2 * 10^4$	10	$1,943 * 10^{1}$s	$2,314 * 10^{1}$s	1,190
$4 * 10^4$	10	$8,341 * 10^{1}$s	$9,859 * 10^{1}$s	1,182
$5 * 10^4$	10	$1,332 * 10^{2}$s	$1,574 * 10^{2}$s	1,180
$1 * 10^5$	10	$5,672 * 10^{2}$s	$6,886 * 10^{2}$s	1,214
$1,5 * 10^5$	10	$1,320 * 10^{3}$s	$1,558 * 10^{3}$s	1,179
$2 * 10^5$	1	$2,400 * 10^{3}$s	$2,837 * 10^{3}$s	1,182
$2,5 * 10^5$	1	$3,827 * 10^{3}$s	$4,514 * 10^{3}$s	1,179

Wie man der Tabelle entnehmen kann, ist unser Code knapp 20% schneller, als der des Compilers. Ursprünglich hatten wir mit einem höheren Zeitgewinn gegenüber des Compilers gerechnet, allerdings hat sich bei genauerem betrachten des Disassembly-Codes gezeigt, dass der Geschwindigkeitsgewinn nicht bereits durch die Verwendung von SIMD-Befehlen zustande kommt.

5.2 Vergleich zum Disassembly-Code

Die größten Geschwindigkeitsunterschiede entstehen durch die Division. Der Compiler verwendet hier die Float-64 Divison. Da wir aber die Genauigkeit der Nachkommastellen nicht benötigen verwenden wir nur 32Bit-Floats. Dies wirkt sich auf die Geschwindigkeit der Konvertierfunktion aus, da der Compiler auch hier die 64Bit Version nutzt. Der Code unterscheidet sich nur in der Struktur und außerdem noch bei dem Vergleich, ob ein Teiler gleich der Zahl selber ist. Hier verwendet der Compiler einen Trick und subtrahiert erst die Zahlen miteinander und zählt dann die vorrangehenden Nullstellen. Danach shiftet er das Ergebnis 5-mal nach rechts. Damit erzeugt der Compiler eine 1, falls die Zahlen identisch sind und sonst eine 0. Wir dagegen vergleichen die zwei Zahlen und arbeiten mit den jeweils gesetzten Flags.

6 Zusammenfassung und Ausblick

Wie bereits in der Einleitung erwähnt scheint der Fokus der Wormell'schen Formel nicht die effiziente Berechnung von Primzahlen zu sein. In dieser Arbeit wurden einige Wege aufgeführt diesen zu optimieren, um wenigstens für kleinere Primzahlen eine akzeptable Performanz zu erreichen. Sowohl das Ersetzen der äußeren Formel durch ein Äquivalent, als auch die Optimierung von Ober- und Untergrenzen, sowie die Parallelisierung und Optimierung in Assembler haben einen enormen Geschwindgkeitszuwachs hervorgebracht. Falls die vorhandene Hardware keine Division oder Parallelisierung anbietet, im Gegensatz zu unserem Szenario, ist es möglicherweise ratsam die Division selbst durchzuführen und feiner an zu nähern. Dies kann ein ähnliches Verfahren nutzen, wie bereits oben gezeigt, jedoch entsprechend mit mehrfacher Wiederholung zur Annäherung der nächst kleineren Stelle.

Für größere n bieten sich zum Beispiel Sieb-Algorithmen, wie das Sieb des Erastosthenes oder das Sieb von Atkin an. Allerdings sind diese recht speicherineffizient und liegen, wie die Wormell'sche Formel selbst auch, in der exponentiellen Wachstumsklasse. Für die zeitgemäße Suche nach Primzahlen bieten sich daher am besten die auf dem AKS-Primzahltest[4] basierenden Methoden an, die sogar eine polynomielle Laufzeit erreichen. Bei den oben genannten handelt es sich zwar lediglich um einfache Primzahltests, doch wie wir durch die Trennung in innere und äußere Formel bereits gezeigt haben, nutzt die Wormell'sche Formel im Kern ebenfalls lediglich einen Primzahltest. Ist dieser Test effizient genug, so ist die Suche nach der n-ten Primzahl trivial.

In der Informatik finden Primzahlen jedoch hauptsächlich eine Anwendung in der Kryptographie. Da bei der benötigten Größe der Primzahlen in diesem Bereich herkömmliche Tests versagen, nutzt man hier so genannte probabilistische Primzahltests, die lediglich zu einer gewissen Wahrscheinlichkeit zeigen, ob eine Zahl prim ist, dafür aber entsprechend performant sind. Die Wormell'sche Formel ist in diesem Bereich aber offensichtlich aufgrund ihrer Laufzeit ungeeignet. Wegen dieser Tatsachen wird die Wormell'sche Formel in Zukunft wohl nach wie vor ein hauptsächlich theoretisches, mathematisches Konstrukt bleiben und keine reelle Anwendungsmöglichkeit finden.

Literatur

[1] Goodstein, R. L. and Wormell, C. P. [*Formulae For Primes*]. The Mathematical Gazette, Band 52, Seiten 35-38, 1967

[2] Sondow, Jonathan and Weisstein, Eric W. [*Bertrand's Postulate*]. Math World – A Wolfram Web Resource

[3] Wolfram Alpha LLC. 2018. Wolfram | Alpha

[4] Weisstein, Eric W. [*AKS Primality Test*]. Math World – A Wolfram Web Resource