Technische Dokumentation KIM-PRKY

Niklas Schütz - 3761908

Aufgabe

Implementieren eines Algorithmus zu Lösung des diskreten Logarithmus.

Der Algorithmus berechnet x, sodass für ein gegebenes g, h und p gilt: g^x = h mod p

Mit der Ausnahme von Shank's Babystep-Giantstep Algorithmus wird außerdem angenommen, dass zwei weitere Variablen n und t bekannt sind. Dabei steht n für die Anzahl an Bits die benötigt werden um x darzustellen (Bitlänge) und t für das Hammington-Gewicht von x.

Algorithmen

Insgesamt wurden die folgenden 5 Algorithmen implementiert

- Shanks Babystep-Gianstep
- dlog_combinations
- dlog_gray_codes
- dlog_shanks_splitting
- dlog_split_combination_arr

Shank's Babystep Giantstep

Der erste Algorithmus der implementiert wurde, ist der Shank's Algorithmus, welcher auch aufgrund seiner Vorgehensweise als Babystep-Giantstep Algorithmus bekannt ist.

Dabei wurde die in der Vorlesung vorgestellte Version implementiert. Beide Versionen haben aber die selbe, im folgenden vorgestellte, Grundidee:

Das x wird aufgeteilt ,sodass gilt `x = x0 + m*x1`
Somit wird aus unseren initialen Formel `g^x modp = h mod p`
die Formel `g^(x0+m*x1) mod p = h mod p`
`Dies wird nun umgestellt sodass gilt:` g^x1^m = h(g^-x0) mod p `

Wir berechnen nun für x1 von 1 bis m `g^m^x1` und speichern die Ergebnisse in einem Dictionary {`g^m^x1` : x1} ab. Das sind die namensgebenden Babysteps.

In den nun folgenden Giantsteps berechnen wir für ein x0 von 1 bis m `h*(g^-x0) mod p` und überprüfen für jedes Ergebnis, ob dieses im Babysteps Dictionary vorkommt. Sollte dies der Fall sein, so haben wir x0 und x1 gefunden für die gilt: `g^(x0+m*x1) mod p = h mod p`

Damit haben wir unser Ergebnis x , wobei `x = x0 + m*x1`

Combinations

Wie bereits erwähnt nehmen wir für die folgenden Algorithmen an, dass die Bitlänge und das Hammington Gewicht von x bekannt ist. Aus dieser Annahme folgt der dlog_combination Algorithmus. Ein Brute-Force Algorithmus, welcher alle x mit der richtigen Anzahl an gesetzen Bits ausprobiert.

Zunächst werden im ersten Schritt alle Möglichen Kombinationen der Länge t aus n berechnet.

Dies erfolgt mit Hilfe des in der Vorlesung gezeigten Algorithmus:

```
def n_ueber_k(n, k):
    """function to calculate all combinations of n over k.
    based on pseudocode given in the lecture
    """
    cur_combination = list(range(k))
    # to not get combinations multiple times we use a set in which we add the
combination as a tuple
    solution_set = {tuple(cur_combination)}
    while 1:
        ok = False
        j = 0
        for j in range(k):
            if cur_combination[k - 1 - j] < (n - 1 - j):
                cur_combination[k - 1 - j] += 1
                solution_set.add(tuple(cur_combination))
                ok = True
                break

        if not ok:
            return solution_set

        j -= 1

        while j >= 0:
            cur_combination[k - 1 - j] = cur_combination[k - 2 - j] + 1
            solution_set.add(tuple(cur_combination))
            j -= 1

    return solution_set
```

Dieser Algorithmus wird ebenfalls in weiteren Algorithmen verwendet. Dort wird allerdings die von Python mit dem itertools Modul bereitgestellte Implementierung `combinations()` verwendet, da diese in C implementiert ist und somit deutliche Performance Vorteile bringt.[1]

Anschließend werden diese Kombinationen, welche in der Form `(1,2,4)` daliegen in Zahlen umgewandelt, wobei jede Zahl der Kombination für die Stelle eines gesetzten Bits steht. `(1,2,3)` steht somit für die Zahl `0b1011` = 11.

```
possible_x = 0
for bit_pos in combination:
    possible_x = possible_x | 1 << bit_pos
```

Zur Erklärung nehmen wir an, dass die aktuelle combination `(1,2,4)` ist.
Zunächst wird eine Zahl possible_x mit dem Wert 0 initialisiert. Danach iterieren wir über jede Bit Position in der Kombination. Wir schieben nun eine 1 bitweise um den Wert der Bitpositione nach links und verodern bitweise unser possible x damit.
Damit erhalten wir am Ende eine Zahl, welche binär an denen in der Kombination genannten Positionen eine 1 hat.

Nun berechnen wir g^x mod p und überprüfen ob das Ergebnis mit h mod p überein stimmt. Ist dies der Fall ist possible_x unser korrektes Ergebnis.

Gray Codes

Dieser Algorithmus implementiert einen anderen Ansatz, mit zwei Besonderheiten die im folgenden erläutert werden.

Anstatt die Kombinationen durchzulaufen laufen wir nur Gray Codes durch. Die Besonderheit von Gray Codes ist, dass sich aufsteigende Codes immer nur um 1 Bit unterscheiden.

Gray Codes für n = 3 Bit:

```
000
001
011
010
110
111
101
100
```

Die zweite Besonderheit ist, dass wir nicht jedes mal g^x mod p berechnen, sondern das vorherige Ergebnis benutzen.
Dazu benötigen wir 2 Arrays, welche im folgenden `set_arr` und `unset_arr` genannt werden. Diese werden wie folgt gefüllt.

Dazu werden der alte und aktuelle Gray Code bitweise verodert um die differenz zu finden. Da sich aufeinanderfolgende Gray Codes um 1 Bit unterscheiden erhalten wir hier immer auch eine Differenz von 1 Bit.

Daraufhin folgt eine Überprüfung ob das unterschiedliche Bit weggefallen oder hinzugekommen ist.

```
diff = code ^ old_code  # get difference
        pos = get_bit_position(bin(diff))
if diff & code:
    code_res = (old_code_res * set_arr[pos]) % p
else:
    code_res = (old_code_res * unset_arr[pos]) % p
```

ist ein Bit hinzugekommen (differenz & code), so multiplizieren wir das alte Ergebnis mit g^(p-1-2^i) mod p, wobei i die Position des hinzugekommenen Bits ist. Da diese Werte sich nicht ändern, können sie im vorhinein berechnet werden und in Arrays gespeichert werden.

unset_arr[i]=g^(2^i) mod p
set_arr[i]=g^(p-1-2^i) mod p

dabei wir `set_arr` benutzt wenn ein Bit gesetzt und `unset_arr` wenn es entfernt werden muss

Sollte das geänderte Bit entfernt worden sein, so wird das alte Ergebnis mit g^(2^i) mod p multipliziert.

BSGS mit Splitting

In diesem Algorithmus gehen wir zurück auf Shank's Babystep Gianststep Algorithmus und verbessern diesen unter Nutzung der Kombinationen.

Dazu splitten wir unser gesuchtes x in zwei Teile x0 und x1, sodass gilt `x=x0+x1`, wobei x0 die Kombinationen von t/2 aus n darstellt und x1 die Restlichen t/2 aus n, für die gilt: Für jede beliebige Kombination k0 aus x0 und beliebige Kombination k1 aus x1 gibt es eine Kombination kn, sodass kn = k0+k1

Allerdings hatte ich mit diesem Splitting Probleme mit meiner Implementierung, sodass ich für x0 und x1 jeweils eine einfach Auswahl t/2 aus n mach. Das führt allerdings dazu, dass ich auch Kombinationen prüfe, die nicht t bits gesetzt haben. Somit kommt es in meiner Implementierung zu einer höheren Laufzeit und Speicherkomplexität.

Aus unseren ursprünglichen Formel `g^x mod p = h mod p` folgt also `g^(x0 +x1) mod p = h mod p`.

Dies lässt sich nun umschreiben in `g^x0 mod p = h * g^-x1 mod p`

Darauf wenden wir nun Shank's Babystep Gianstep Idee an, wobei wir allerdings nicht wie bei Shanks √p Möglichkeiten durchlaufen müssen, sondern nur n über t/2 Möglichkeiten.

Wir speichern also unsere `g^x0 mod p` in einem Babystep Dictionary.

In den Giantsteps rechnen wir nun für jede Kombination aus x1 den rechten Term (`h * g^-x1 mod p`) aus und überprüfen ob das Ergebnis sich in den Babysteps befindet. Sollte dies der Fall sein, so haben wir unser x gefunden. x kann nun aus den beiden zueinander passenden Kombinationen errechnet werden.

Combinations mit Splitting und Berechnung durch Set/Unset Arrays

Im letzten und finalen Algorithmus werden nun alle bisherigen Ideen kombiniert.

Die erste Idee ist die Nutzung der Set und Unset Arrays, die im Gray Code Algorithmus eingeführt wurden.

Am Anfang entstand die Idee, dass wir die Kombinationen durchlaufen und das Ergebnis "normal" mit g^x mod p neu berechnen vorher aber prüfen, ob die Unterschiede zwischen den Kombinationen so gering sind, dass wir besser die Arrays benutzen. Dort stellte sich allerdings herraus, dass es in den meisten Fällen besser ist direkt nur die Arrays zu nutzen.

Diese Idee wurde nun mit dem vorherigen Algorithmus verbunden.

Wir splitten die Kombinations also auf und berechnen mit einem Teil unsere Babysteps und mit dem anderen unsere Giantsteps. Dabei nutzen wir das vorherige Ergebnis und multiplizieren je nach Bitunterschieden der Kombinationen mit den entsprechenden Arrays.

Benchmarks

Rahmenbedingung

Für die folgenden Benchmarks und Vergleiche zwischen den unterschiedlichen Algorithmen gelten die folgenden Rahmenbedingungen:

CPU	Ryzen 5 1600[^2] (übertaktet auf 3,7GHZ)
RAM	16GB @ 3000 Mhz
Pagefile/SWAP size	25 GB
Python Interpreter	Python 3.10.0 (tags/v3.10.0:b494f59)

[^2] https://en.wikichip.org/wiki/amd/ryzen_5/1600

Zusätzlich sind einige Werte über alle Tests hinweg gleich:
Für p wurde die 64 bit lange Primzahl 12355317818073449027 gewählt
Für g wurde 187 gewählt.

Für jede Bitlänge von x wurde die zu testende Funktion 4 mal mit zufälligen Werten aufgerufen. Zur Erstellung einer zufälligen Zahl mit n gesetzen Bits wurde dafür die Funktion `x = random.getrandbits(n)` genutzt.

Timing Decorator

Um die Tests zu timen wurde ein Decorator `@timing_logging_decorator` implementiert.

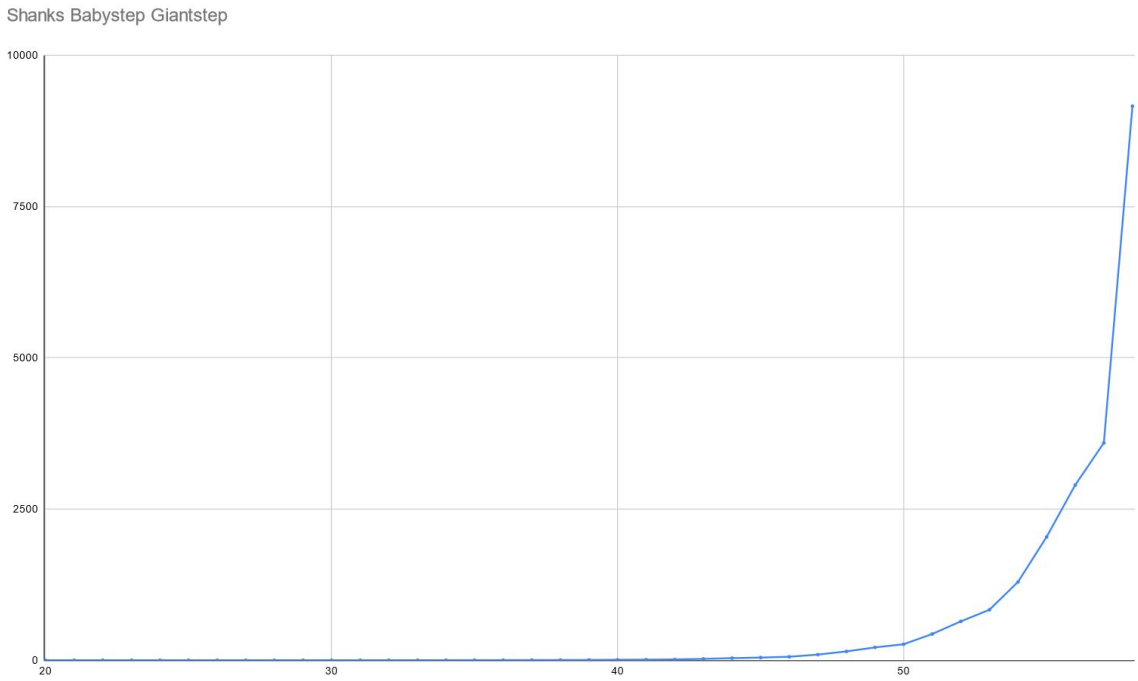
```
def timing_logging_decorator(fnc):
    """decorator to time the function and log the result to stdout"""

    global x_bit_length

    if "x_bit_length" not in globals():
        x_bit_length = "no bitlength given as global value"

    def inner(*args, **kwargs):
        start = perf_counter()
        result = fnc(*args, **kwargs)
        end = perf_counter()

        time = float(format(end - start, ".5f"))

        log(
            f"{fnc.__name__}: {x_bit_length} bit in {time} seconds",
            file=fnc.__name__,
        )

        return result, time

    return inner
```

Diese führt die Funktion aus und misst, wie lange die Funktion braucht. Dabei wird Die Zeit in Sekunden gemessen und auf 5 Stellen nach dem Komma begrenzt.

Ebenfalls wird eine eigene Funktion `log()` aufgerufen, Diese erhält als Parameter den String und optional einen Dateinamen. Sollte der file Parameter gesetzt werden, so wird der übergebene String neben dem Standartoutput auch in eine Datei geloggt.

Shank's Algorithmus

Im ersten Schritt wurde Shanks Algorithmus bis zu einer Länge von 58 bit getestet:



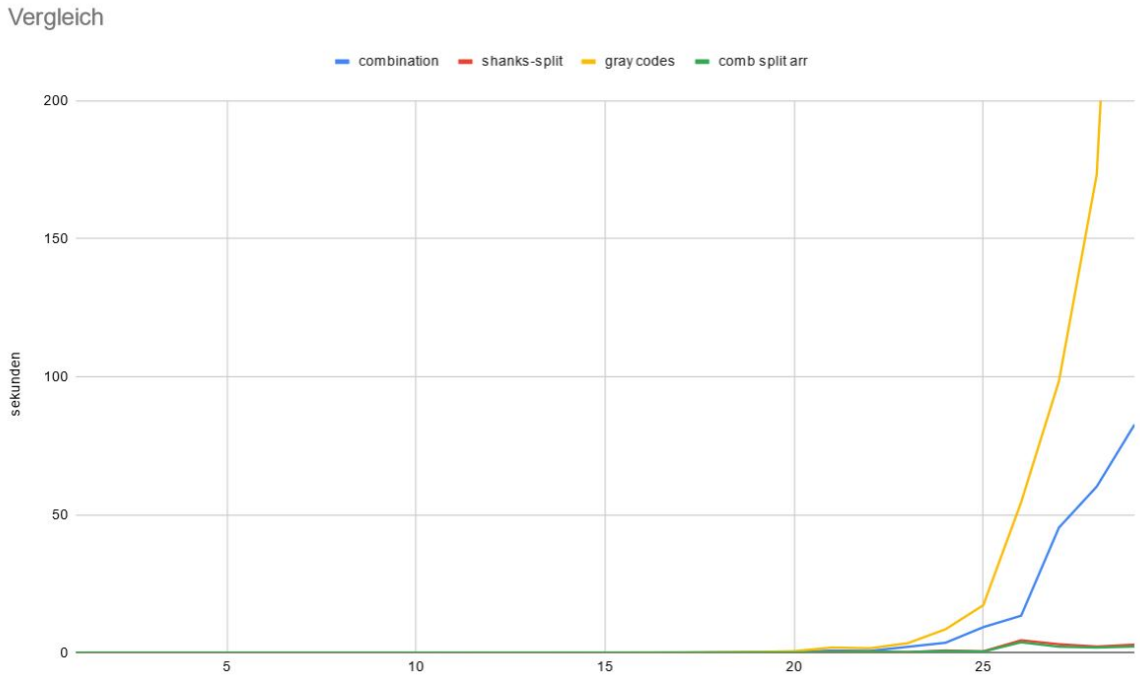
Wie im Graph zusehen ist, ist Shank's Algorithmus bis 40 bit (7.47625 Sekunden) noch effizient. Allerdings steigt die benötigte Zeit exponentiell, bis sie den Höhepunkt von 9156 Sekunden bei 58 Bit erreicht.

Vergleich der Algorithmen mit gegebener Bitlänge und Hammington Gewicht

Im folgenden wurden die Algorithmen getestet, die vorraussetzen, dass die Bitlänge von x und die Anzahl der gesetzten Bits gegeben ist.

Dazu wurden für die Bitlängen 1 bis 29 getestet. Für jede Bitlänge wurde mit je 4 zufälligen Zahlen jeder der Algorithmen gebenchmarked.

Aus Gründen der Übersichtlichkeit wurde der Anzeigebereich des Diagramms auf der Vertikalenachse auf 200 sekunden beschränkt. Für 29 bit benötigt der Gray-Code Algorithmus allerdings 433 Sekunden und damit deutlich länger als alle anderen.



Vergleich der Algorithmen mit gesplitteten Combinations mit Babystep Giantstep

Da im ersten Test die beiden Algorithmen sehr ähnliche Zeite hatten wurde erneut ein Test für die Algorithmen `Shanks mit Split` und `Combinations mit Splitting und Berechnung durch Set/Unset Arrays` durchgeführt, um zu prüfen ob die Nutzung der Hilfsarrays einen Vorteil bringt. Dabei wurde 5 mal ein zufälliges x mit 32 Bit ausgewählt und beide Algorithmen damit gebenchmarked. Wie bereites im Test davor wurde als p die 64 Bit lange Zahl 12355317818073449027 und als g 187 gewählt.

Algorithmus	Lauf 1	Lauf 2	Lauf 3	Lauf 4	Lauf 5	Durchschnitt	Std.abweichung
Shanks mit Split	46.656	73.49518	131.87722	214.74378	39.55028	101.26449	39.55028
Combination mit Split und Array	29.96691	53.99258	88.67004	154.5768	29.96921	71.43511	52.32795

Durch den Test fällt auf, dass die Berechnung mit Hilfe der Arrays bei 32 Bit etwa 70% der Zeit braucht, die eine normale Berechnung benötigt. Allerdings haben wir dort eine etwas höhe Standartabweichung.

Fazit

Sind die Bitlänge und das Hammington Gewicht von x bekannt, so können wir einen deutlichen Performance Vorteil erhalten, indem wir nur die infragekommenden Kombinationen testen. Durch das Splitting der Kombinationen in Verbindung mit den Baby und Giantsteps lässt sich die Anzahl der nötigen Berechnungen weiterhin verringern.

Zu guter Letzt wurden die Berechnungen effizienter implementiert, indem wir anstelle einer arbeitsaufwendigen Neuberechnung das alte Ergebnis nutzen und je nach (un-)gesetztem Bit lediglich eine einfache Multiplikation durchführen. Betrachtet man die Implementierung in Python so fällt auf, dass diese in anderen Programmiersprachen mit einem bessren Support für Operationen auf Bit-Level noch effizienter implementiert werden können.

1. https://github.com/python/cpython/blob/main/Modules/itertoolsmodule.c