Kilka słów o optymalizacji na przykładzie

Podczas czwartego spotkania zaimplementowaliśmy bardzo proste "liniowe" rozwiązanie problemu naprawy pliku, który został uszkodzony w trakcie ściągania z naszego serwera po bardzo wolnym łączu. Nasze rozwiązanie polegało na podzieleniu pliku (1GB) na równe fragmenty (1MB), a następnie przesłaniu 256-bitowych (32-bajtowych) haszy wszystkich fragmentów – jeden po drugim – z naszego serwera na nasz komputer i sprawdzeniu które hasze zdalne są różne od lokalnych. Jeśli któraś para haszy była różna, to – na żądanie klienta – serwer przesyłał cały jedno megabajtowy fragment danych.

Rozwiązanie testowaliśmy na pliku, który był uszkodzony w 5ciu miejscach. Każde uszkodzenie było wielkości 1 bajtu, tj. w sumie 5 losowych bajtów w pliku było zmienione na nieprawidłową wartość. Wszystkie dalsze rozważania w tym materiale będą się opierać właśnie na pliku uszkodzonym w ten sposób.

Policzmy na szybko ile danych zostało przesłanych w naszej implementacji, aby naprawić ten plik:

- 1GB podzielony na fragmenty po 1MB daje 1024 fragmenty.
- Dla każdego fragmentu serwer musiał przesłać jego hash (SHA256) w formie binarnej, tj. 32 bajty. To daje nam **32768** bajty (32KB) na hasze.
- Klient odpowiadał "OK" (hasze się zgadzają) lub "UP" (hasze się nie zgadzają, prześlij cały fragment). Czyli po 2 bajty na każdy fragment, w sumie **2048** bajtów (2KB).
- 5 fragmentów było uszkodzonych, co daje nam 5MB przesłanych danych.
- **W sumie**: 32KB + 2KB + 5MB, czyli **5 277 696 bajtów**.

Nie jest źle, ale może być lepiej. Czy moglibyśmy to więc **zoptymalizować**?

Dygresja o optymalizacji

Optymalizowanie działania kodu jest jedną z najbardziej satysfakcjonujących czynności w programowaniu. Jednocześnie – jak wszystko w programowaniu – jest to temat rzeka. Poświęcę więc chwilkę na ogólne przedstawienie problematyki optymalizacji.

Po pierwsze trzeba wskazać, że optymalizuje się "pod coś", tj. pod jakąś konkretną cechę, tudzież jakiś konkretny pomiar. Przykładowo, program taki jak nasz można optymalizować pod kątem:

- ... całkowitego czasu wykonania (krócej == lepiej);
- ... czasu przesyłu danych (krócej == lepiej, ale liczy się tylko czas transferu);
- ... czasu użytego przez procesor (krócej == lepiej, ale nie liczymy czasu kiedy wątek "śpi", tj. na coś czeka);
- ... zużycia pamięci RAM (mniej == lepiej);
- ... wielkości kodu (mniej == lepiej; patrz: code golfing);
- ... wielkości kodu bajtowego po kompilacji (mniej == lepiej; patrz `python -m compileall`)
- ... prostoty implementacji (łatwiej == lepiej);
- ... wykorzystania dysku (mniej == lepiej);
- i tak dalej.

Optymalizacja to często jest tzw. "trade-off" ("coś za coś", tj. kompromis między dwoma – lub większą liczbą – cech). Np. możemy "zapłacić" większym wykorzystaniem procesora za mniejsze zużycie pamięci RAM. Albo w drugą stronę – możemy "zapłacić" większym wykorzystaniem pamięci RAM w zamian za krótsze wykonanie programu.

Od razu zaznaczę, że koncept "płacenia" tutaj nie odbywa się w sposób bezpośredni, a jest raczej efektem ubocznym metod (algorytmów i ich parametrów) które wybieramy. Tj. dla danego problemu mogą istnieć metody (algorytmy), które są mniej intensywne od strony CPU ale zużywają więcej RAM, i mogą istnieć metody, które zużywają mniej RAM, ale za to więcej czasu CPU. Mogą, ale nie muszą.

Jeśli chodzi o podejścia programistyczne do optymalizacji, to w sumie są dwa. Dodam, że nie jest to oficjalna klasyfikacja, a bardziej sposób, w który ja osobiście na to patrzę:

1. **Optymalizacja na poziomie implementacji**, która polega na dbaniu o to, by wybrana przez nas metoda była napisana w jak najlepszy (pod względem wybranych kryteriów) sposób. W tym podejściu bierze się pod uwagę to jak działa sprzęt, system operacyjny, czy sam język programowania i jego środowisko wykonania.

2. **Optymalizacja na poziomie algorytmiki**, która polega na dobraniu odpowiedniej metody (algorytmu) dla rozwiązania danego problemu. Zazwyczaj jest to kwestia wymiany całego algorytmu na jakiś lepszy (pod względem wybranych kryteriów) lub większych modyfikacji w kontekście wariantu danego algorytmu.

Granica między tymi podejściami jest dość rozmyta, ale na szczęście sama granica nie jest za bardzo istotna. Co natomiast jest istotne, to, że optymalizacja na poziomie algorytmiki jest

zazwyczaj dużo skuteczniejsza dla większej liczby danych, ponieważ pozwala przejść np. z rozwiązania liniowego (formalnie: O(n) – patrz "notacja wielkie O") na rozwiązanie np. logarytmiczne (formalnie: O(logn)). Tłumacząc "na ludzki" oznacza to, że jeśli w pierwotnym podejściu "liniowym" dla przetworzenia 1000 fragmentów pliku musielibyśmy wykonać 1000 operacji (ew. jakiś inny iloczyn liczby operacji, tj. K * 1000), to w podejściu "logarytmicznym" byłoby to już tylko około 10 operacji. Z kolei dla 2000 fragmentów w podejściu "liniowym" byłoby to 2000 operacji, ale w "logarytmicznym" już tylko 11 – to wynika z faktu, że funkcja logarytmiczna rośnie dużo dużo wolniej, niż funkcja liniowa.

Optymalizacja na poziomie implementacji nie zmieni zależności pomiędzy ilością danych a liczbą operacji, które trzeba wykonać. To co może natomiast zmienić, to wspomniany wcześniej czynnik K, czyli np. ile cykli procesora faktycznie zajmie wykonanie pojedynczej operacji.

Co za tym idzie, najkorzystniej jest najpierw wybrać najlepszy znany algorytm dla rozwiązania danego problemu (optymalizacja na poziomie algorytmiki), a następnie upewnić się, że jest optymalnie zaimplementowany (optymalizacja na poziomie implementacji).

Ale... nie tak szybko!

I teraz przechodzimy do bardzo istotnego faktu o optymalizacji, którego bardziej początkujący programiści **bardzo nie lubią słyszeć**.

Otóż... **optymalizacja często zupełnie nie ma sensu**.

Przykładowo, załóżmy, że mamy pewien program, o którym wiemy, że przetwarza milion zestawów danych w ciągu 10 minut. Czyli trochę mu to zajmuje – aż się prosi o zoptymalizowanie!

Ale, zanim rzucimy się do optymalizacji, powinniśmy popatrzeć na to jak ten program jest faktycznie – w rzeczywistym świecie – wykorzystywany, a także jaki byłby koszt optymalizacji.

Może się na przykład okazać, że w ciągu roku program przetwarza tylko 6 milionów zestawów danych, tj. chodzi 60 minut. Jednocześnie, optymalizacja zajęłaby nam – od przejrzenia kodu i zlokalizowania tego co tam jest takie wolne[1], przez wymyślenie optymalnego rozwiązania, jego implementację tego rozwiązania, i zdebugowanie wszystkich problemów, do porządnego przetestowania i wdrożenia nowego rozwiązania – około 100 godzin roboczych. Prosta matematyka mówi, że program nie zużyje tyle czasu co nasze "optymalizowanie" przez 100 lat. Co więcej, w przypadku firmy czy innej instytucji, koszt 100 roboczogodzin czasu programisty jest bardzo wymierny i spokojnie mógłby zjeść wszystkie potencjalne zyski.

Co za tym idzie, w tym przypadku lepszy rozwiązaniem mogłoby być po prostu nic nie robienie. Ewentualnie można by rozważyć wynajęcie super szybkiego serwera na tę godzinę, przez którą program przetwarza dane – na pewno wyszło by to dużo taniej niż 100 godzin czasu programisty, w szczególności, że komputery nadal robią się z roku na rok coraz szybsze.

[1] Jak konkretnie to zrobić jest kolejnym tematem-rzeką, ale sprowadza się do wykonywania pomiarów, które pozwalają ustalić np. co zużywa za dużo zasobu, pod kątem którego chcemy zoptymalizować nasz program.

To nie wyklucza oczywiście istnienia przypadków, w których jak najbardziej warto poświęcić czas na optymalizację. Warto w głowie mieć często cytowaną (niestety zazwyczaj tylko częściowo) maksymę autorstwa Donalda Knutha:

> *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"*

Oczywiście podczas nauki programowania, jak i dla zabawy (słynne hakerskie *for fun & profit*), możemy optymalizować co tylko chcemy :)

P.S. Skoro jesteśmy przy temacie optymalizacji, to wspomnę jeszcze z kronikarskiego obowiązku, że fraza "bardziej optymalne [np. rozwiązanie]" jest kością niezgody pomiędzy językowymi purystami a resztą programistów. Tj. formalnie program może "być optymalny" albo "nie być optymalny" – czyli "bycie optymalnym" jest binarne (jest albo nie jest) i nie jest skalą od "mniej optymalnych" do "bardziej optymalnych". W codziennym slangu programistycznym programiści często jednak mówią, że dane rozwiązanie jest "bardziej optymalne" albo "mniej optymalne", ponieważ i tak wszyscy wiedzą o co chodzi.

Optymalizacja naprawienia pliku

Po tej długiej dygresji wróćmy jednak do naszego problemu naprawy pliku 1GB uszkodzonego w pięciu miejscach.

Załóżmy, że w celach edukacyjnych (tudzież dla zabawy), chcemy zoptymalizować naprawę pliku **pod kątem całkowitego czasu jej trwania** (tzw. *wall time*, tj. czasu według zwykłego zegara na ścianie).

Wiemy, że większość czasu schodzi na przesłanie danych, bo nasze przykładowe łącze ma prędkość 256kbit/s, czyli niecałe[2] 32KB/s. Co za tym idzie, możemy skupić się właśnie na liczbie przesyłanych bajtów.

Nasz punkt wyjściowy to wspomniane na początku 5 277 696 bajtów, czyli około 161 sekund na sam transfer danych (odczyt danych z pliku i liczenia haszy możemy póki co zignorować, bo to dużo mniejsze wartości, więc trudno by tam było coś znaczącego "ugrać").

Patrząc na problem od strony optymalizacji na poziomie implementacji, to co możemy w łatwy sposób kontrolować, to wielkość fragmentu. Obecnie fragmenty mają 1MB, co sprawia, że większość przesyłanych danych to właśnie poprawne fragmenty danych.

A gdyby tak zmniejszyć większość fragmentu?

[2] Niecałe, bo 256kbit/s to pełna przepustowość, natomiast trochę bajtów również jest "zużywane" do przesyłania różnego rodzaju metadanych, w szczególności w protokołach niższych warstw sieciowych. Więc efektycznie nie jest to pełne 32KB/s, tylko trochę mniej – zapewne coś około 30KB/s.

Jest to bardzo dobry pomysł, ale musimy też wziąć pod uwagę, że mniejszy pojedynczy fragment sprawia, że zwiększa się całkowita liczba fragmentów – a więc i liczba haszy, i "OK" / "UP" które muszą być przesłane. Znajdźmy więc (numerycznie, bo to kurs programowania a nie matematyki) optymalne rozwiązanie:

```
print(
    f"{'frag_sz':7}  {'total_hash_sz':>14}  {'total_meta':>14}  {'total':>14}"
)
FILE_SZ = 1 * 1024**3
for n in range(5, 20):
  frag_sz = 2**n
  frag_count = FILE_SZ // frag_sz
  total_hash_sz = frag_count * 32  # 32 bytes per hash.
  total_meta = frag_count * 2  # "OK" or "UP" for each hash.
  total = 5 * frag_sz + total_hash_sz + total_meta

  print(f"{frag_sz:7}  {total_hash_sz:14}  {total_meta:14}  {total:14}")
```

Powyższy program wypisze nam wszystkie potrzebne dla nas informacje jeśli chodzi o zależność wielkości fragmentu oraz liczby przesłanych bajtów. Oto wynik jego działania.

frag_sz	total_hash_sz	total_meta	total
32	1073741824	67108864	1140850848
64	536870912	33554432	570425664
128	268435456	16777216	285213312
256	134217728	8388608	142607616
512	67108864	4194304	71305728
1024	33554432	2097152	35656704
2048	16777216	1048576	17836032
4096	8388608	524288	8933376
8192	4194304	262144	4497408
16384	2097152	131072	2310144
32768	1048576	65536	1277952
65536	**524288**	**32768**	**884736**
131072	262144	16384	933888
262144	131072	8192	1449984
524288	65536	4096	2691072

Jak od razu można zauważyć, optymalną wielkością fragmentu wydaje się być 65536 (tudzież jakaś liczba leżąca w okolicy tej wartości), dla której musielibyśmy przesłać 884736 bajtów, czyli poniżej 1MB! Moglibyśmy delikatnie przerobić ten program, żeby znalazł dokładne minimum lokalne dla całkowitej liczby przesłanych bajtów – znalazłby on wtedy wielkość fragmentu 85421, co przełożyłoby się na około (bo z dokładnością do zaokrągleń) **854451 bajtów**.

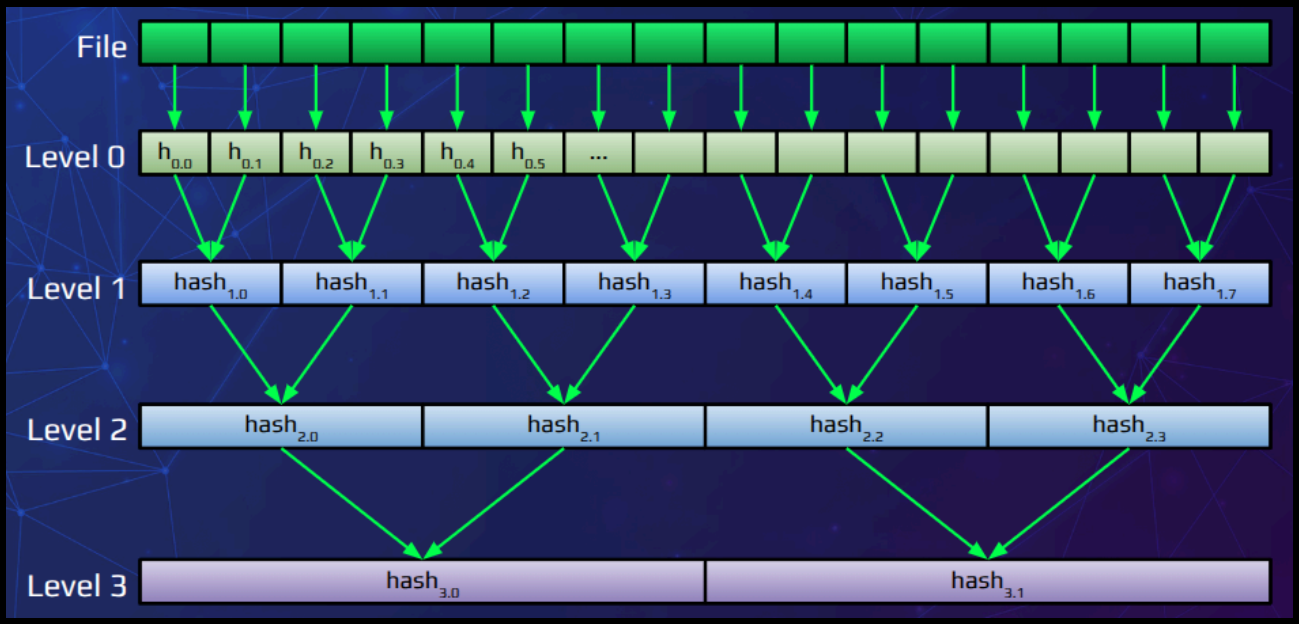
Co za tym idzie, sama wielkość fragmentu zmieniłaby czas ze 161 sekund, do 26 sekund potrzebnych na transfer.

Ale niestety w tym momencie dotarliśmy prawie do granicy możliwości jeśli chodzi o możliwości optymalizacji pod względem implementacji. Owszem, można by jeszcze spróbować:

- ... przesyłać krótsze hasze, np. tylko 16 bajtów zamiast 32, co by zbiło czas o kolejne kilka sekund. Co istotne, czym bardziej schodzimy z wielkością hasza, tym bardziej rośnie prawdopodobieństwo trafienia w kolizję (popsuty fragment i dobry fragment dający ten sam hasz). 16 bajtów nadal jest bezpieczne i pewnie można by zejść jeszcze o kilka bajtów zanim problem zacząłby się objawiać w praktyce (ale jak już się objawi, to weźmie nas z zaskoczenia).
- ... zamiast wysyłać "OK" albo "UP" wystarczy wysłać jedną literkę. Ba! Można by wysyłać mapę bitową, w której zapalone są tylko bity odpowiadające fragmentom do przesłania. Ale tutaj możemy ugrać może z 1 sekundę.

Zapewne dałoby się jeszcze coś wymyślić na tym poziomie, ale ostatecznie zeszlibyśmy do około 10 sekund (i to ryzykując kolizje haszy).

Optymalizacja na poziomie algorytmiki



Spróbujmy zatem zmienić podejście.

Pierwszy etap pozostawimy bez zmian – plik podzielimy na fragmenty i z każdego fragmentu wyliczymy jego hasz. Tak otrzymaną listę haszy nazwiemy "poziomem 0" haszy.

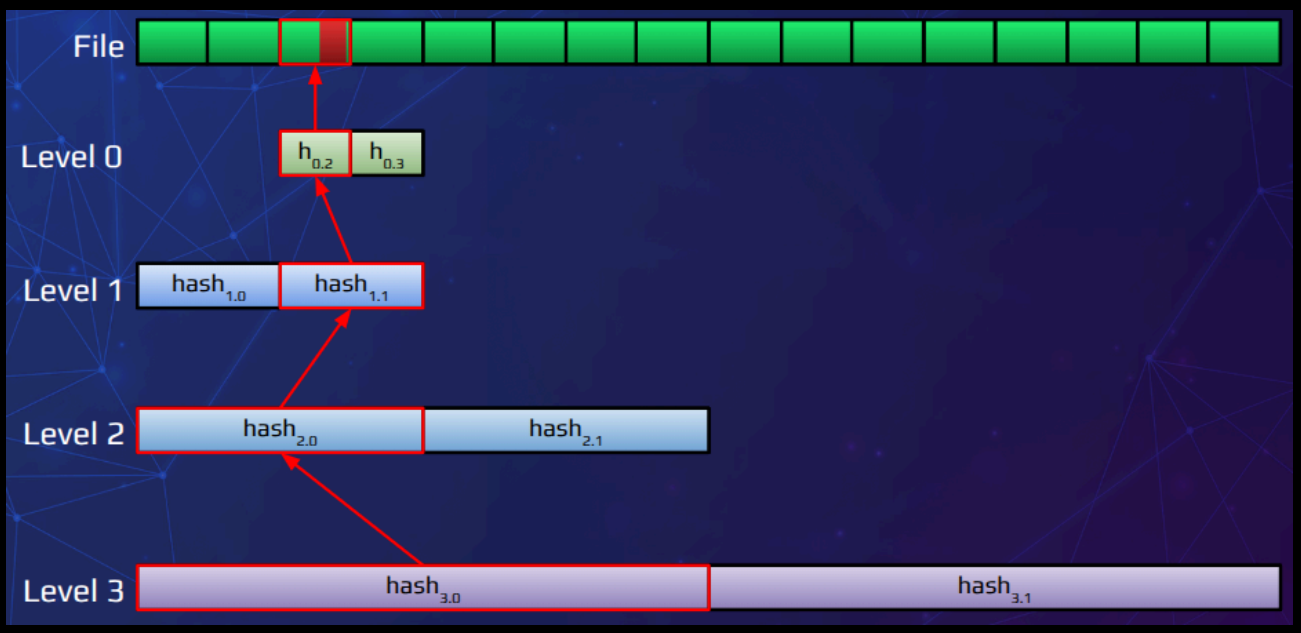
Następnie weźmiemy każde kolejne pary haszy (np. hasz 0 i 1, hasz 2 i 3, itd.) i je "połączymy", tj. wyliczymy nowy hash "poziomu 1" z pierwszego hasza do którego dokleimy drugi hasz[3].

[3] Można takie "łączenie" haszy zrobić na kilka różnych sposobów. Wyliczanie nowych hashy ze sklejonych hashy jest jednym z wolniejszych rozwiązań (tj. zużywa więcej czasu CPU), ale zapewnia trochę lepszą wykrywalność błędów niż np. zXORowanie dwóch haszy.

Na każdym poziomie proces ten będziemy powtarzać dla wszystkich par haszy, a poziomy będziemy dodawać do póki nie dojdziemy do poziomu, na którym zostaną tylko 2 hasze (patrz ilustracja na poprzedniej stronie).

Musimy jeszcze zmienić podejście klienta. Tj. klient powinien móc zapytać o konkretny hasz na danym poziomie, np. hash 45 na poziomie 7. Dzięki temu, klient będzie mógł zacząć od ostatniego poziomu i zapytać o oba hasze. Jeśli oba hasze są zgodne z tym co klient wyliczył lokalnie – super, to oznacza, że plik nie jest uszkodzony.

Załóżmy jednak, że pierwszy hasz z pary się różnił. Wiemy, że hasz ten odpowiada za dwa "pod hasze", tj. jeden lub oba te "pod hasze" musiały być różne. Co za tym idzie, klient może teraz zapytać o oba te hasze na kolejnym poziomie i zobaczyć który z nich się różnił. Zakładając, że mamy tylko jeden uszkodzony bajt w tym regionie, klient będzie musiał ostatecznie zapytać co najwyżej o dwa razy więcej haszy niż liczba poziomów – a poziomów będzie około logarytm z liczby fragmentów o podstawie dwa, czyli np. 10 dla 1000 fragmentów, lub 20 dla 1 miliona fragmentów – niespecjalnie dużo. Patrz również ilustracja poniżej:



W przypadku naszego przykładowego pliki 1GB uszkodzonego w 5 miejscach, dla fragmentów o wielkości 1KB (czyli mielibyśmy ponad 1 milion fragmentów) musielibyśmy wymienić jedynie około 100 haszy oraz 5KB samych danych. To daje około 11 KB danych, a więc na transfer potrzeba jedyne 0.35 sekundy!

Gdzie jest haczyk?

Otóż tworzenie "drzewa" haszy trwa wymierną ilość czasu. Czym mniejszy fragment, tym więcej haszy trzeba wyliczyć. Co więcej, czym więcej haszy trzeba przechować, tym więcej potrzeba pamięci RAM. Przykładowo, w nieco skrajnym przypadku, dla fragmentów o wielkości 64 bajty drzewo na moim laptopie tworzy się około 30 sekund i zużywa niecałe 3GB pamięci RAM.

Co za tym idzie, ostatnim krokiem jest dobranie rozsądnej wielkości fragmentu, tak, żeby zbalansować czas tworzenia drzewa oraz czas samego transferu danych.

Poniższa tabelka przedstawia przykładowe wyniki dla dla różnych wielkości fragmentów:

Legenda:

- FRAG_SZ – wielkość fragmentu
- TOTAL_RECV – całkowita liczba odebranych bajtów przez klienta (a więc wysłanych bajtów przez serwer)
- TOTAL_SENT – całkowita liczba wysłanych bajtów przez klienta (a więc odebranych przez serwer)
- TOTAL – całkowita liczba bajtów przesłanych pomiędzy stronami
- CPU – zmierzony czas procesora potrzebny na przygotowanie drzewa (ta wartość zazwyczaj jest obarczona pewnym błędem, ale nie wpłynie na wyniki w naszym przyapdku)
- NET – szacunkowy czas przesyłu danych między stronami na łączu 256kbit/sec
- WALL – całkowity czas naprawy pliku 1GB dla pięciu 1-bajtowych uszkodzeń

FRAG_SZ	TOTAL_RECV	TOTAL_SENT	TOTAL	CPU	NET	WALL
32	7476	1165	8641	55.6s	0.26s	55.8s
64	7188	1095	8283	30.6s	0.25s	30.9s
128	7124	1035	8159	15.5s	0.24s	15.8s
256	7124	935	8059	8.5s	0.24s	8.7s
512	8660	975	9635	5.2s	0.29s	5.5s
1024	10836	915	11751	3.5s	0.35s	3.8s
2048	15636	865	16501	2.7s	0.50s	3.2s
4096	**25300**	**775**	**26075**	**2.2s**	**0.79s**	**3.0s**
8192	45908	795	46703	1.8s	1.32s	3.1s

Jak można odczytać z tabeli powyżej, o ile najmniej danych zostało wysłane dla fragmentów o wielkość 256 bajtów, to – z uwagi na 8 i pół sekundy potrzebnych na utworzenie drzewa haszy – nie jest to najkorzystniejszy przypadek. W praktyce bardziej korzystna okazała się być wielkość fragmentu ustawiona na 4096 bajtów (4KB), która ładnie zbalansowałą czas tworzenia drzewa (2.2s) oraz czas przesyłu danych (0.79s), dając ostatecznie czas 3.0s.

Dodam, że wszystkie pomiary powyżej zostały wykonane dla 32-bajtowych haszy, więc nadal mamy tutaj miejsce do optymalizacji na poziomie implementacji.

Podsumowanie

Podsumowując, w tym przypadku z pierwotnych 161 sekund udało się zejść do 3. Całkiem niezły wynik, będący w dużej mierze zasługą lepszego – ale i bardziej skomplikowanego w implementacji – algorytmu.

Bardzo zachęcam, żeby prędzej czy później oprócz programowania zainteresować się również Algorytmiką – taką pisaną z wielkiej litery. O ile w praktyce rzadko kiedy implementuje się lub

wymyśla się jakieś bardziej skomplikowane algorytmy podczas codziennego programowania, to jak przyjdzie co do czego, wiedza ta staje się bardzo przydatna[4].

A wracając jeszcze do naszej zoptymalizowanej metody: zachęcam Was do zastanowienia się, która metoda byłaby lepsza (pod względem całkowitego czasu wykonania), jeśli w pliku 1GB co drugi fragment (niezależnie od wybranej wielkości) zawierałby uszkodzony bajt.

Gynvael Coldwind

P.S. Zoptymalizowaną implementację można znaleźć tutaj:
https://cdn-hackarcana.net/py10-1dc19979d3098465305bb2df6781b461/py10-day04-additional-01.zip

[4] Alternatywnie trzeba wiedzieć w którym pokoju w biurze siedzi jakiś dobry Algorytmik ;)