Algoritmus

Jáchym Löwenhöffer

Gynekologická Evaluace Velkých Obrazů

jachym.lowenhoffer@gmail.com

February 12, 2025

Přehled

Logaritmus algoritmus v těle mám... rytmus

Co to je?

Formální definice algoritmu je složitější, ale intuitivní úplně stačí:

Algoritmus

Posloupnost dostatečně jednoduchých kroků s proveditelných kroků, které řeší nějaký problém v konečném čase.

V těch "dostatečně jednoduchých krocích" je problém, protože hodně záleží na tom kdo bude ten příslušný algoritmus číst a vyhodnocovat.
Příklady:

- Recept v kuchařce
- Navigace v telefonu
- Program v Pythonu
- Čistá binárka
- Návod na složení nábytku

Kdo to čte?

Pro počítače píšeme jiné algoritmy než pro lidi.

V obou případech se ale algoritmus **překládá** než se reálně vyhodnotí (až třeba na binárku ta už se rovnou spustí). Musíme brát v potaz kdo nebo co náš algoritmus překládá abychom ho napsali dobře.

Na chybu při kompilování (překládání do jednodušších instrukcí) u algoritmů pro lidi často narazíme až pozdě nebo dokonce nikdy.

Co znamená dobrý algoritmus?

Algoritmy potřebujeme porovnávat. Kritérií může být hodně, zkuste vymyslet to **nejlepší**. Nejintuitivnějším kritériem je prostě absolutní čas za který program doběhne. Ten je v praxi samozřejmě důležitý, ale je závislý na parametrech stroje a dalších faktorech, což nechceme.

Proto pro účely teoretické informatiky měříme rychlost algoritmus vůči velikosti vstupu. Tedy když dostane n čísel na vstupu a sečte je dohromady tak provedl $\mathcal{O}(n)$ operací.

Důležité je u výpočtu časové složitosti pamatovat, že uvažujeme ten nejhorší případ možný. Je programátorovou úlohou hledat ty nejhorší vstupy a z nich určit asymptotickou časovou složitost (neboli $\mathcal{O}$).

Definice $\mathcal{O}$

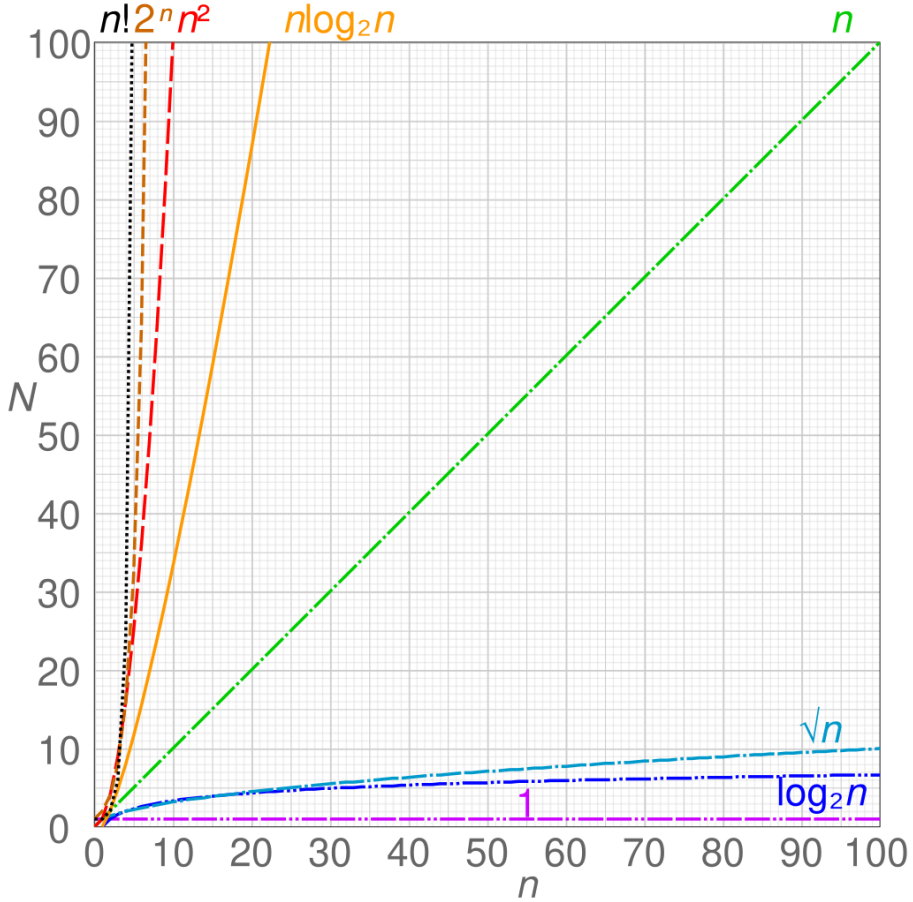
Jak bude jasné z definice níže, symbol $\mathcal{O}$ se používá jako horní odhad nějakého algoritmu. Je tedy pravda že program na sčítání provedl taky $\mathcal{O}(n^{n!})$ kroků. Ale samozřejmě se snažíme najít co nejmenší horní hranici.

Technická definice $\mathcal{O}$

Do třídy $\mathcal{O}(g)$ patří všechny funkce f pro které existuje $c \in \mathbb{R}$ takové že $f \leq cg$ platí až na konečně mnoho výjimek.

Ukázky mainstream $\mathcal{O}$

Když uvádíme rychlost našeho algoritmu často co nejvíc zjednodušujeme výraz kterým ho odhadujeme. To vedlo až k tomu že se vytvořili nějaké nejpoužívanější třídy algoritmů podle časové složitosti.



A co prostorová složitost?

Krom toho jak dlouho algoritmus běží je občas taky důležité kolik místa v paměti zabere. Tuto velikost taky uvádíme relativně k velikosti vstupu.

Moderní programovací jazyky za nás ty očividné prasárny s pamětí zpravují ale při vymýšlení algoritmu stojí za letmou úvahu.

Příklad

Časovou složitost si ukážeme na třídících algoritmech. Představte si, že jste dostali neseřízený seznam hodnot které mezi sebou dokážete porovnávat a vy je chcete seřadit od nejmenší po největší. Jak na to? možností je víc:

- **Miracle sort** - zkouší opravdu každou možnost $\mathcal{O}(n!)$.
- **Buble sort** - vezme každý prvek a porovná ho s každým ještě nesrovnaným tedy v nejhorším případě $\mathcal{O}(n^2)$.
- **Quicksort** - podle prostředního prvku intervalu setřídí každou půlku a rekurzivně pokračuje v každé půlce - $\mathcal{O}(n^2)$.

Co to asi bude?

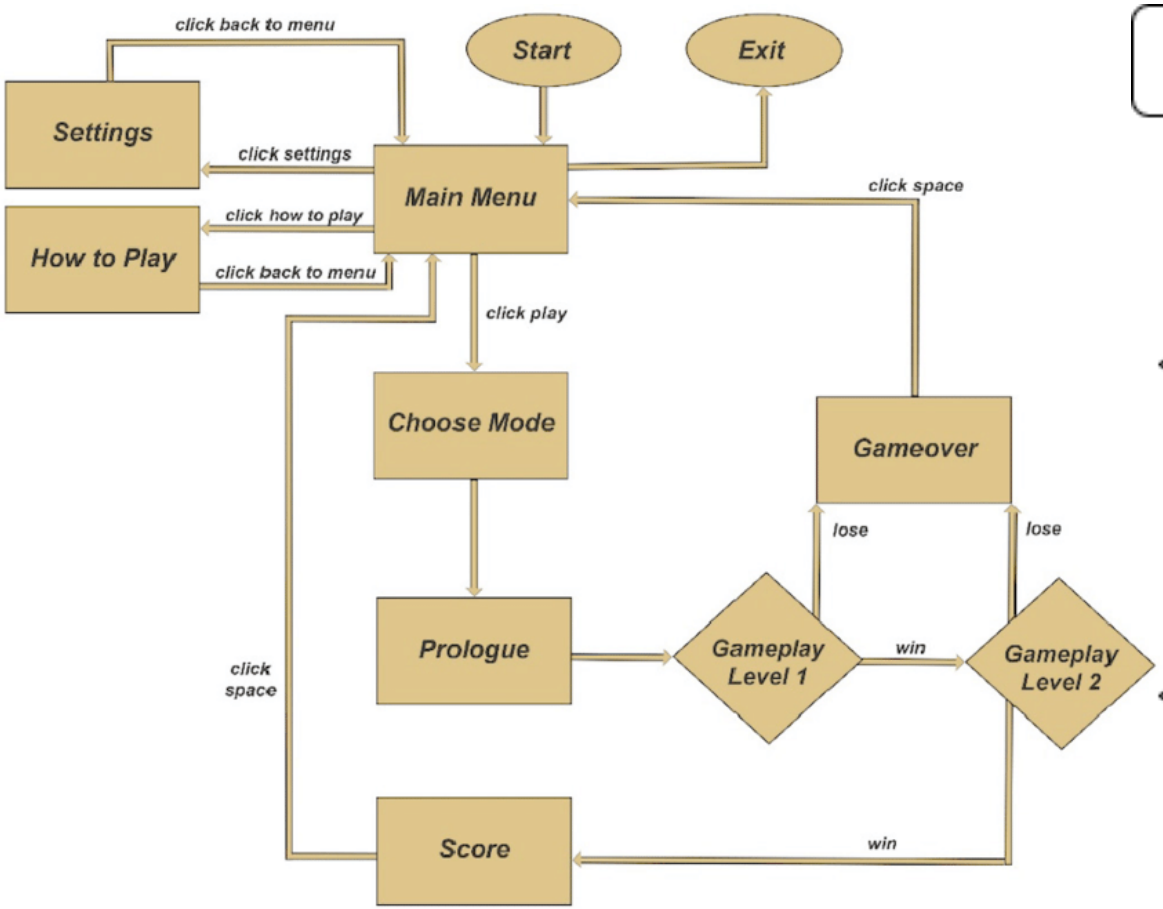
Bro co viděl mojí sick flow byl jako: "tpc to si musím načrtnout".

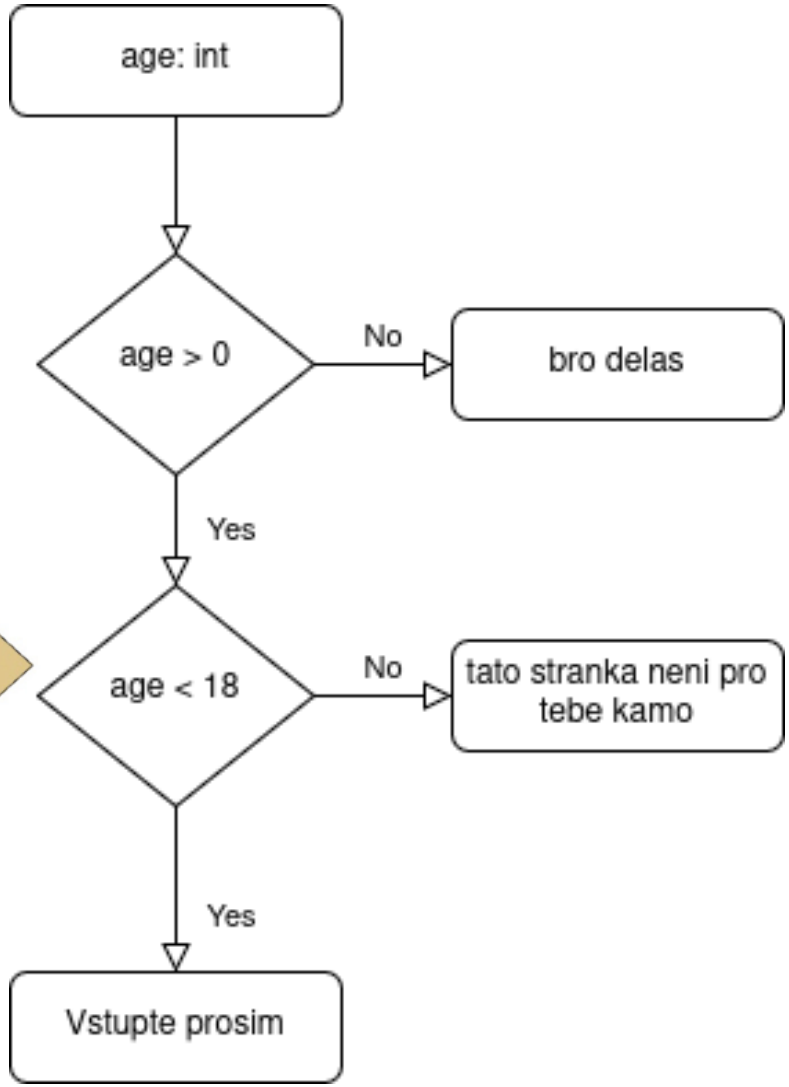
Flowchart

Jde o způsob jak zjednodušit komplikovanější strukturu tak aby byli jasně vidět které kroky na sobě závisí a kde se různé průchody můžou větvit na základě nějaké podmínky.

Flowcharty se používají při designování algoritmů. Je jen na nás s jakou přesností je budeme dělat.

Příklady dvou přesností flowchartů:





Převod mezi do kódu

Když už si uděláme flowchart toho jak náš program bude fungovat, tak teď je na řadě samotné programování. Kód pro kontrolu věku z minulého slidu můžeme naprogramovat například:

```
age = input("Zadej věk")

if age < 0:
    print("bro delas")
if age < 18:
    print("tato stranka neni pro tebe kamo")
else:
    print("Vstupte prosim")
```

Programátorské techniky

Rekurze

Věc které se děje v programování docela často je, že jedna funkce volá samu sebe. To by na flowchartu vypadalo tak, že z jednoho políčka vede šipka do toho samého políčka. Důležité je mít na paměti, že rekurze musí mít nějakou podmínku při které se zastaví jinak nikdy nedoběhne.

Ideální příklad je hledání Fibonacciho čísel. Když dostaneme na vstupu n tak to definujeme jako součet té stejné funkce pro vstupy $n-1$ a $n-2$. Přičemž musíme dodefinovat $F(0)=0$ a $F(1)=1$.

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

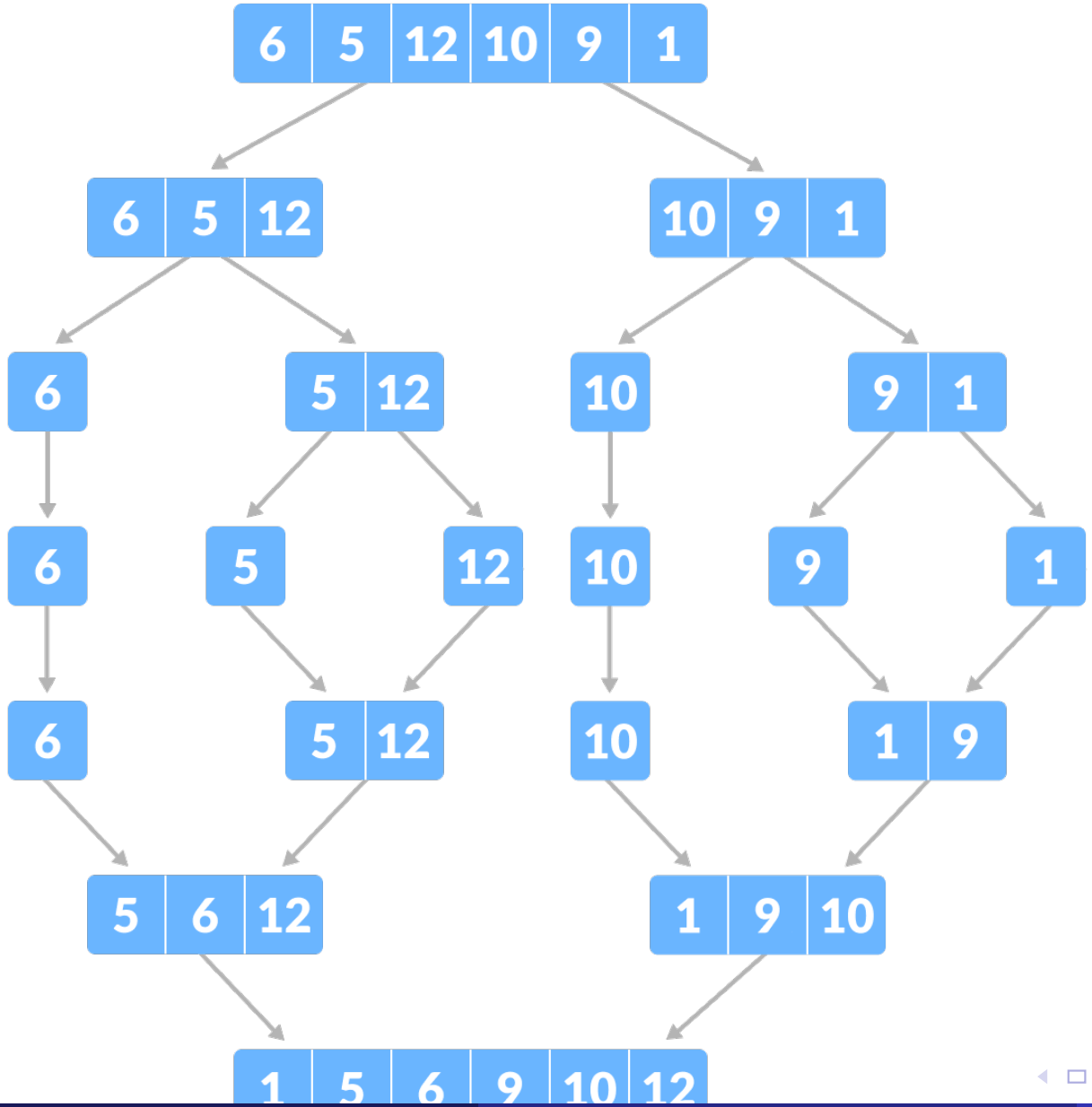
Nejlepší část rekurze je, že ji z pravidla nechceme a ani nemusíme používat, často se dá nahradit while loopem. Zkuste to pro Fibonacciho čísla.

Rozděl a panuj

Omg tento problém je tak těžký, co kdybych si ho rozdělil na spoustu malých částí, které už dokážu vyřešit jednoduše a potom dal všechny ty řešení dohromady? To by ale znamenalo že jsem problém rozdělil a panoval. :sunglass emoji:

A protože neexistují jiné algoritmu než ty řadící ukážeme si to na merge sortu:
Místo abychom seřadili velký pole tak si to rozdělíme až dojdeme do jednotlivých elementů které jsou ofc každý seřazený a potom vždy spojíme dvě sousední pole tak aby výsledek zůstal seřazení což je ez.

Ukázka merge sortu



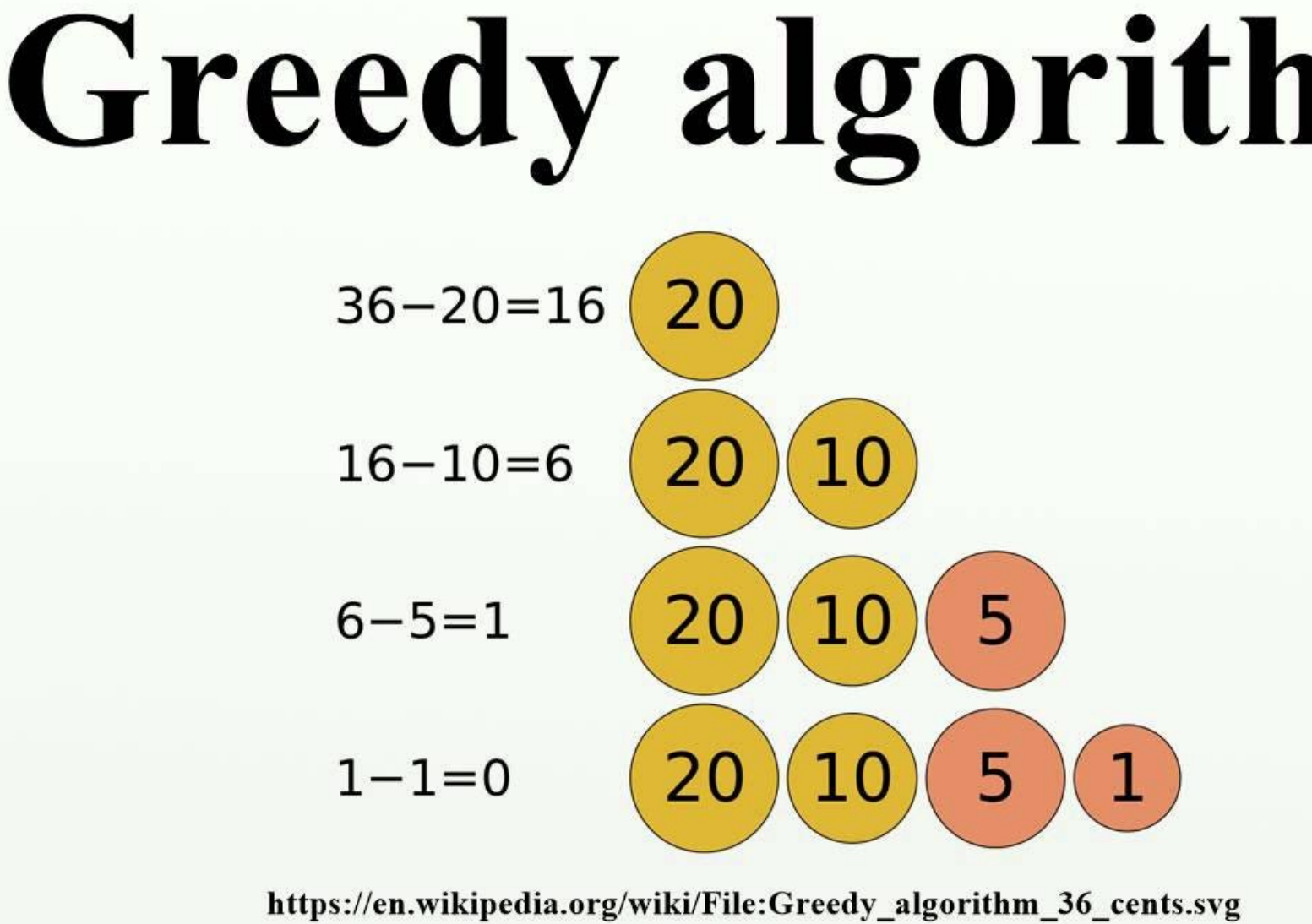
Hladovost

Co když řešení komplexního a složitého problému jako celku je na mě prostě moc? Co kdybych prostě v každém kroku mohl vybrat to, co zni nejlíp a doufal že se dostanu do cíle?

Třeba když chci najít nejkratší cestu z A do B, tak prostě můžu vzít tu nejkratší ulici na každé křižovatce žejo?

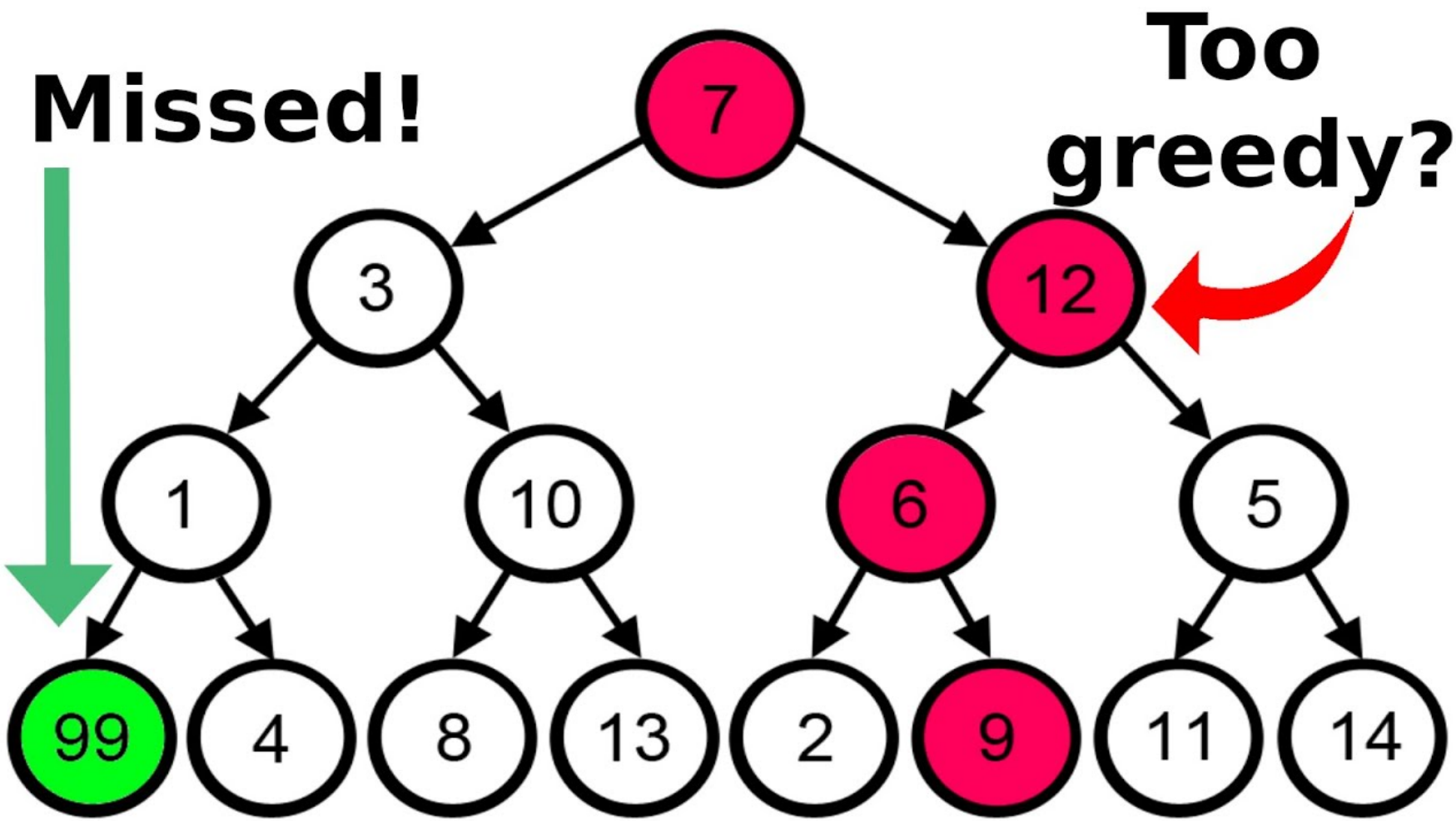
Příklad této myšlenky, jak funguje je například vracení peněz. Dostaneme částku kterou máme vydat v mincích. Najdeme největší minci která je menší než daná částka, tu vypíšeme a spustíme funkci znova s částka - mince (nemusí vždy fungovat, ale naše mince jsou postavené aby to fungovalo). Další příklad je Kruskal.

Obrázky



https://en.wikipedia.org/wiki/File:Greedy_algorithm_36_cents.svg

Dalsi obrazky



Backtracking

Backtracking je jen hezké slovo pro brute-force. Nejlépe je myšlenka "vracení se" vidět na procházení bludiště. Obecná myšlenka je, že procházíme nějaký stavový strom do hloubky.

Jdeme nějakou chodbou dokud nenarazíme na rozcestí nebo na konec chodby. Jestliže se jedná o konec vrátíme se na nejbližší rozcestí a zkusíme jinou chodbu. Jestliže dorazíme k rozcestí spustíme funkci znova na náhodné chodbě a poznačíme si která to je (nepočítáme s existencí cyklů).

Dynamické programování

Zakládá se hlavně na tom, že si před spuštěním samotného algoritmu předpočítáme nějaké hodnoty a ty nám poté o hodně zrychlí zbytek programu. Samozřejmě musíme řešit jestli se to ve výsledku vyplatí. Když chceme najít podposloupnost s největším součtem posloupnosti:

$$1, -2, 4, 5, -1, -5, 2, 7$$

Tak si uděláme seznam S který bude mít na i-tém místě součet hodnot 0 až i z původního seznamu. Pak ale součet podposloupnosti z a do b je $S[b] - S[a]$.

Celková časová složitost je jen $\mathcal{O}(n^2)$ což je o dost lepší než jen naivní řešení které je $\mathcal{O}(n^3)$ (funguje úplně stejně jen součet daného úseku naivně počítá).