テキスト処理 第12回 (2008-07-08)

田中哲
産業技術総合研究所
情報技術研究部門
akr@isc.senshu-u.ac.jp
http://staff.aist.go.jp/tanaka-akira/textprocess-2008/

今日の内容

- レポートの解説
- 正規表現エンジンの停止性
- 正規表現エンジンの計算量
- レポート

停止性

- 以下では無限再帰防止が必要だった
  try([:rep, [:empstr]], ["a"], 0, {}) {}

- 無限再帰防止をしない場合
  - スタックが溢れて止まる
  - Segmentation Fault などの異常終了

再帰を停止させる方法

- 再帰呼出しするごとにに少しでも処理を進める
- 処理が有限であれば終わる

正規表現エンジンの処理の進み

- 処理が進む:
  - 残りの文字列が短くなる
  - パターンが小さくなる
- 再帰するたびに処理が進めば、無限には再帰しない
  - 文字列の長さは有限
  - パターンの大きさは有限

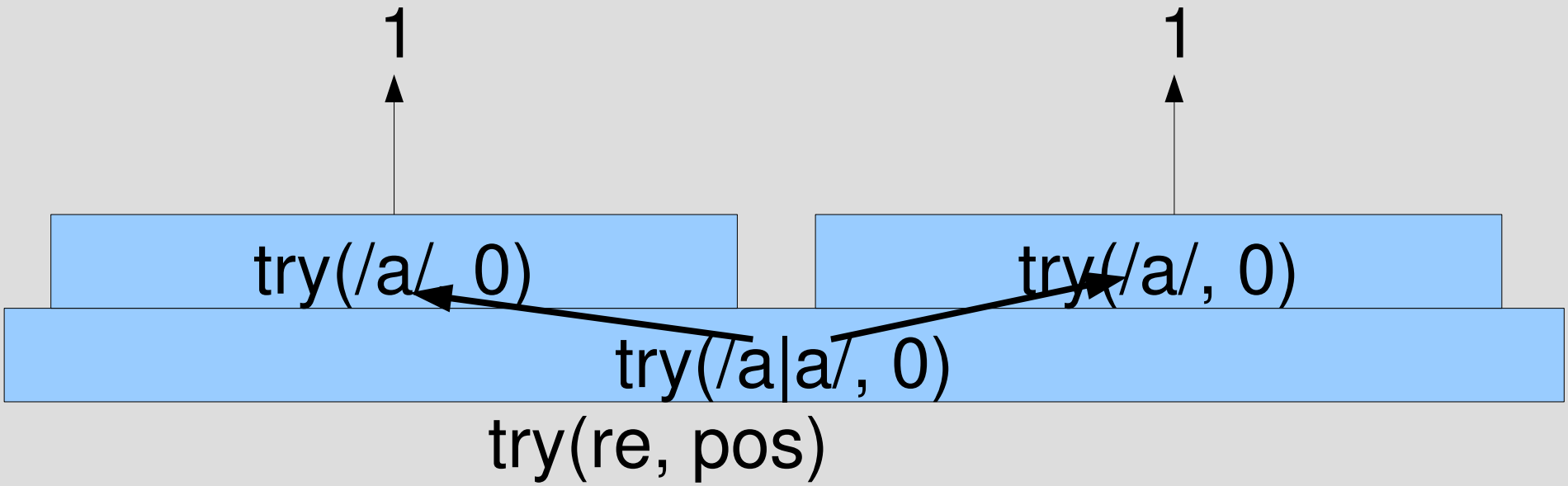
try_alt

- [:alt, r1, r2] より r1 と r2 は小さい
  r1, r2 は [:alt, r1, r2] の一部分
- try([:alt, r1, r2]) は try(r1) と try(r2) を呼び出すのでパターンが小さくなっている
- 文字列の残り (pos 以降) は変わらない

```
def try_alt(re, str, pos, md, &b)
  try(re[1], str, pos, md, &b)
  try(re[2], str, pos, md, &b)
end
```

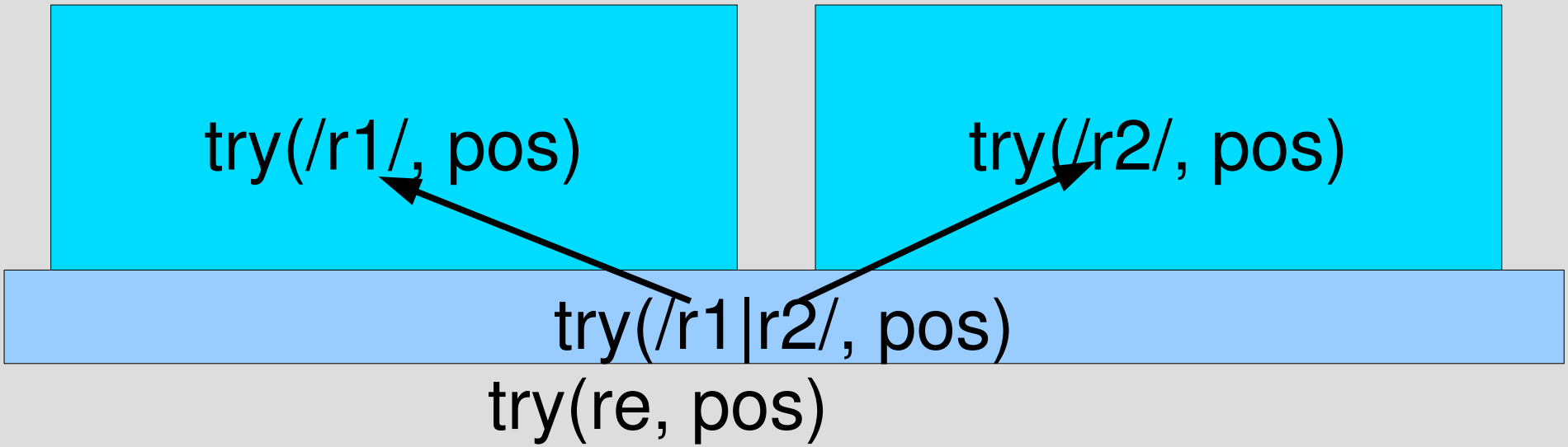
簡単のため2引数版を考える

/a|a/ =~ "a"



try_alt を再帰呼出しすると正規表現が小さくなる

/r1|r2/ =~ str



- try_alt を再帰呼出しすると正規表現が小さくなる
- try(r1) は小さくなってるので再帰は有限
- try(r2) も小さくなっているので再帰は有限
- どちらも再帰は有限

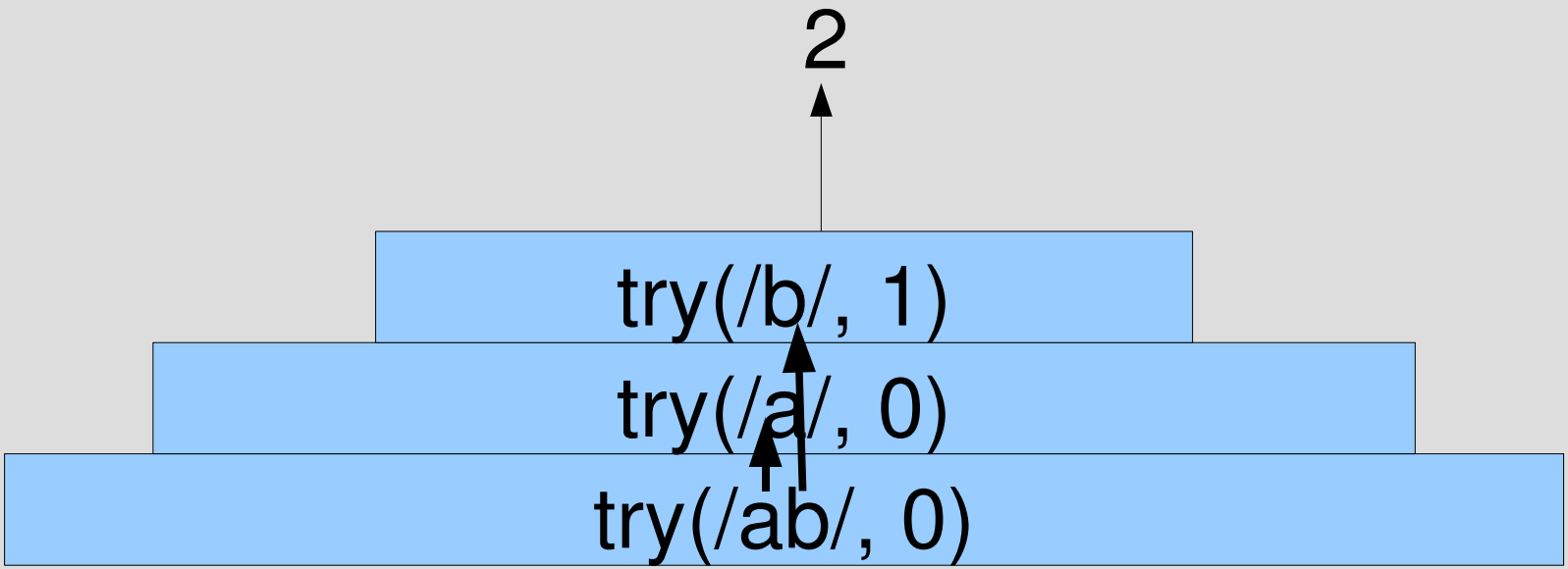
try_cat

- [:cat, r1, r2] より r1 と r2 は小さい
- try([:cat, r1, r2]) は try(r1) と try(r2) を呼び出し、そのときパターンは小さくなっている
- 文字列の残りは、増えることはない
  (try(r2) については減るかもしれない)

```
def try_cat(re, str, pos, md, &b)
  try(re[1], str, pos, md) {|pos2, md2|
    try(re[2], str, pos2, md2, &b)
  }
end
```

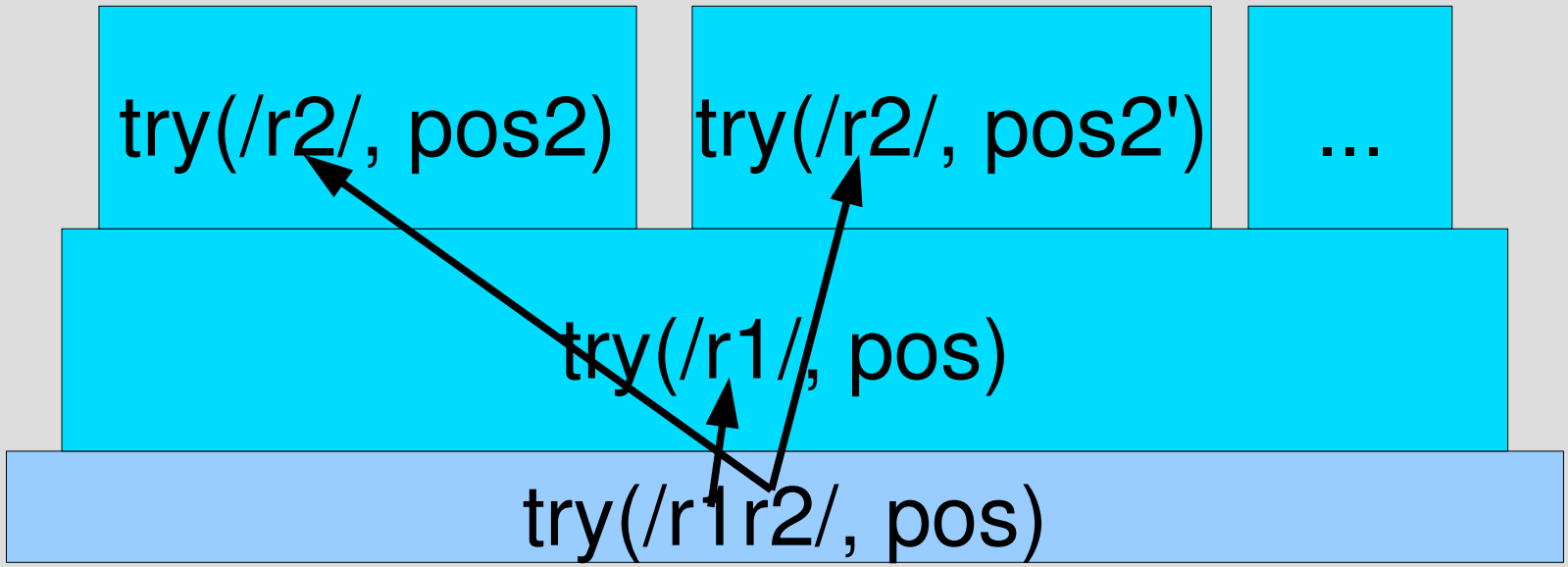
簡単のため2引数版を考える

/ab/ =~ "abc"



try_cat を再帰呼出しするとパターンが小さくなる

/r1r2/ =~ str



- try_cat を再帰呼出しするとパターンが小さくなる
- try(r1) は小さくなってるので再帰は有限
- try(r2) も小さくなってるので再帰は有限
- 有限をふたつ足しても有限
- 結局、再帰は有限

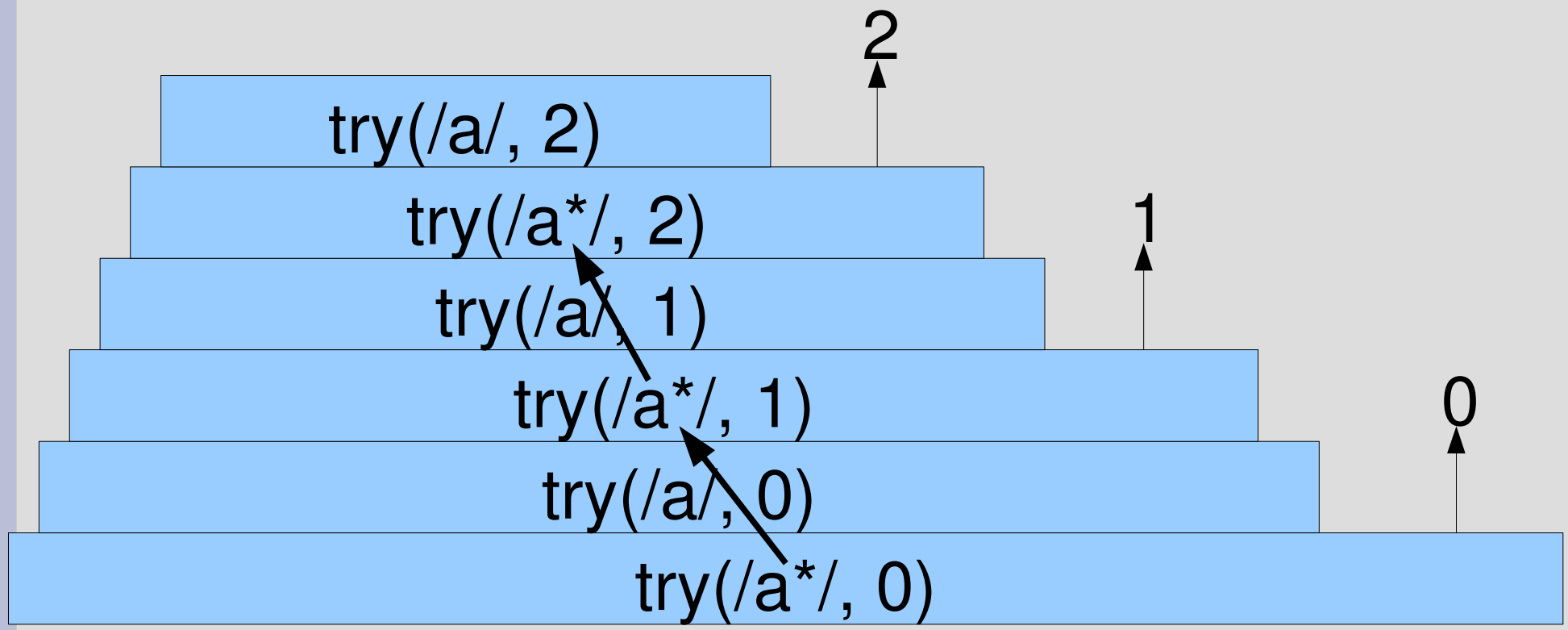
try_rep

- [:rep, r] より r のほうが小さい
- try([:rep, r]) は try(r) を呼び、小さくなってる
- try_rep([:rep, r]) は try_rep([:rep, r]) を呼ぶがこれは同じ大きさ
- でも pos < pos2 なときしか再帰しない

```
def try_rep(re, str, pos, md, &b)
  try(re[1], str, pos, md) {|pos2, md2|
    try_rep(re, str, pos2, md2, &b) if pos < pos2
  }
  yield pos
end
```

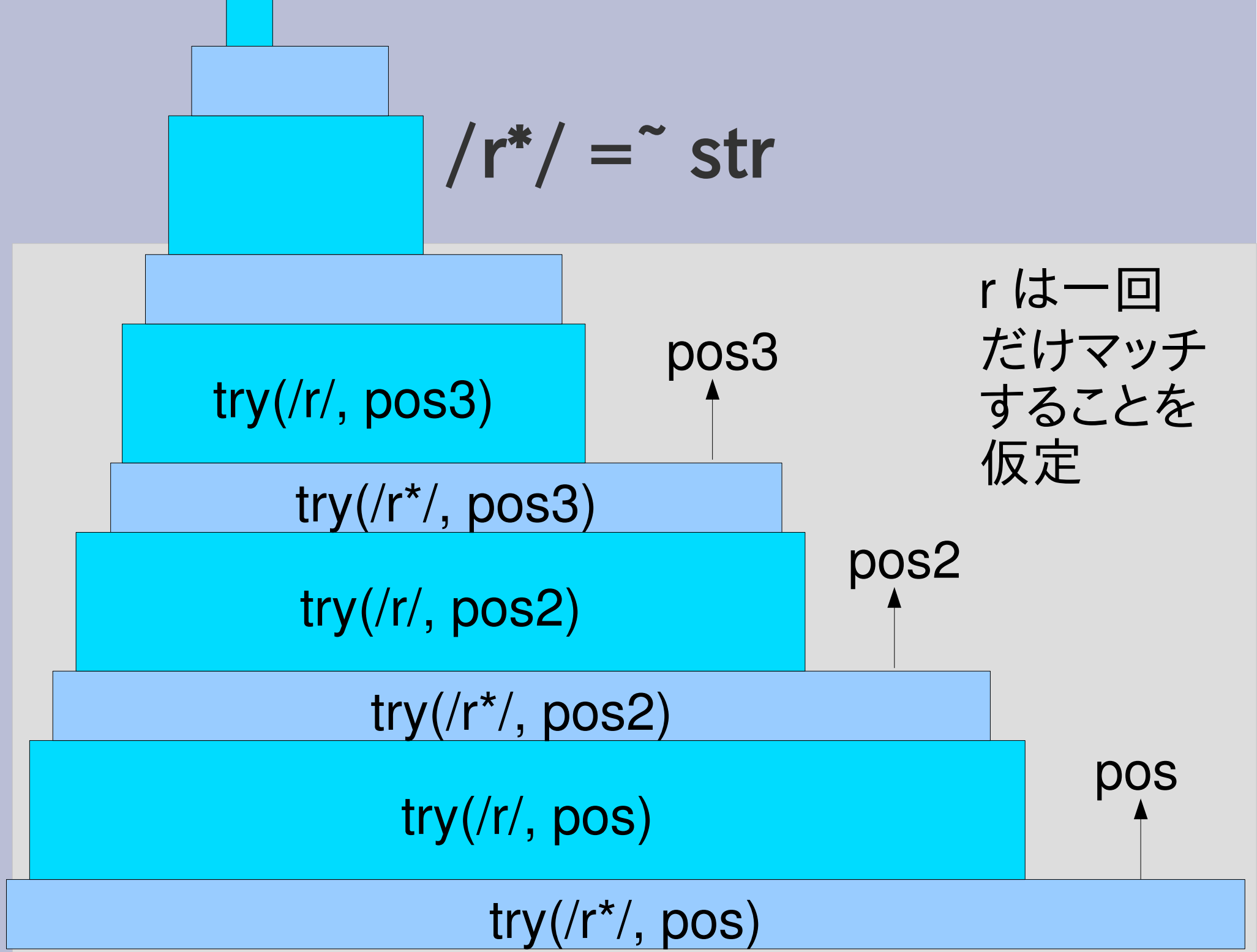
/a*/ =~ "aa"

- try(/a*/) が何回も重なる
- でも pos は増えていくので有限



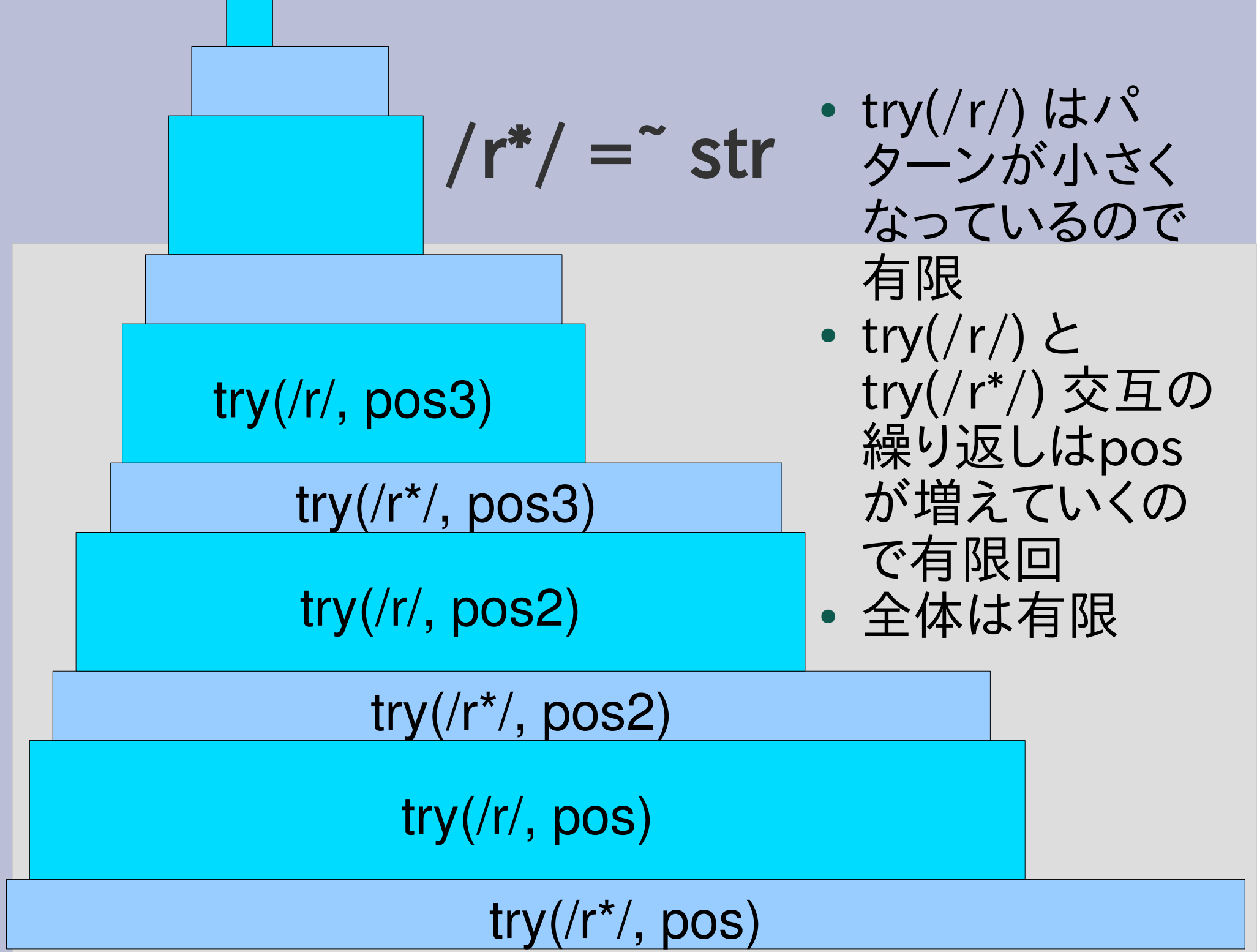
/a*/ =~ "aa"

- try(/a*/) はいくらでも重なる
- でも毎回 pos が進む
- ただし、pos を進めるのは try(/a/)


/r*/ =~ str
try(/r/, pos3)
try(/r*/, pos3)
try(/r/, pos2)
try(/r*/, pos2)
try(/r/, pos)
try(/r*/, pos)
pos3
pos2
pos
r は一回だけマッチすることを仮定

try(/r*/)と無限再帰

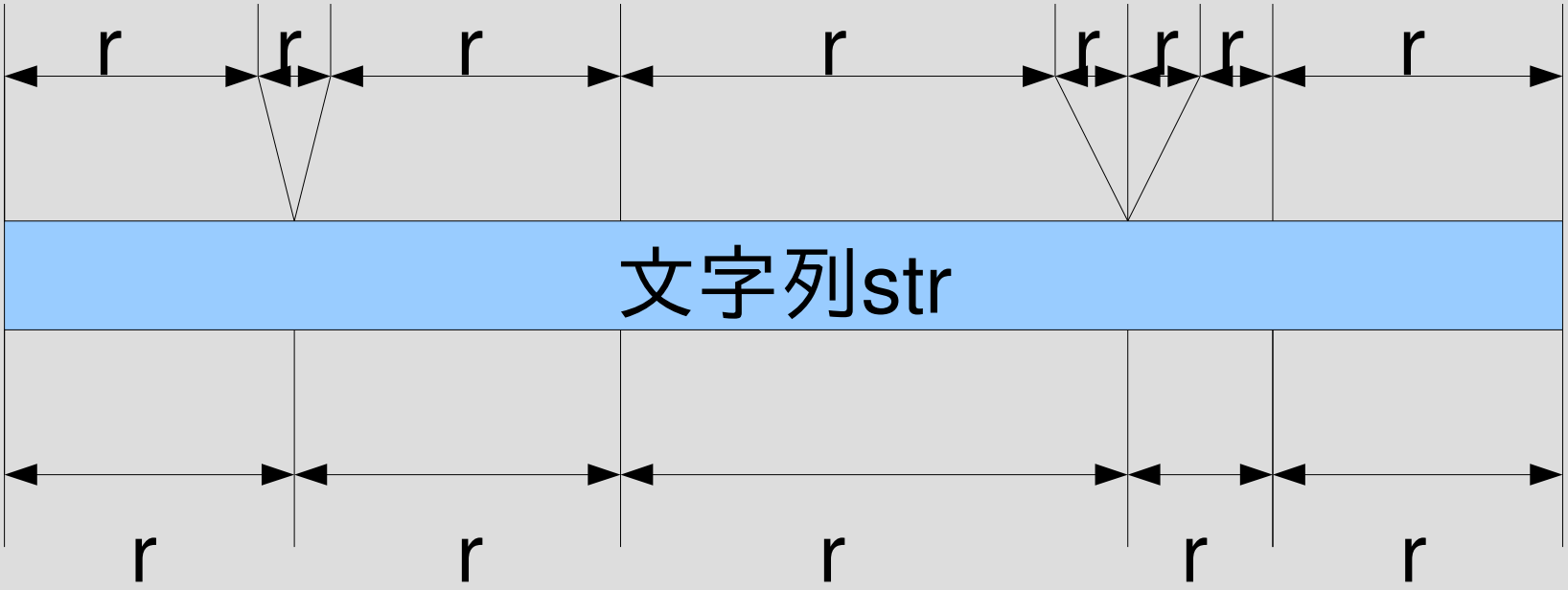
- try(/r*/) が try(/r*/) を呼び出す
- パターンが小さくならない
  - これはいつも成り立つ
- 残りの文字列
  - r が空文字列にマッチしたら pos == pos2 になる
  - 例えば /()*/ とか
    抽象構文木だと [:rep, [:empstr]]
  - でもそのときは再帰しない (無限再帰防止)
    再帰するのは pos < pos2 のときだけ
- 再帰するときは常にパターンか残りの文字列が小さくなる



- try(/r/) はパターンが小さくなっているので有限
- try(/r/) と try(/r*/) 交互の繰り返しはposが増えていくので有限回
- 全体は有限

無視してもマッチするものはマッチする

/r*/ =~ str



空文字列へのマッチを取り除いて、
全体へのマッチを作れる
空文字列を無視しても
マッチしなくなるわけではない

停止性

- 再帰の深さは有限
- ループも有限
  - 任意個引数 try_alt は正規表現が与えられれば最大ループ回数は決まる
- そのうち終わる

計算量

- 有限時間で終わることを保証できた
- では、どのくらい時間がかかるか？
  - 今日中に終わる?
  - 太陽の寿命が尽きるまでに終わる?
- try は何回呼び出されるか？

try の呼び出しを数える

```
def count_try(re, str)
  $try_count = 0
  rx_ends(re, str, 0)
  $try_count
end

def try(re, str, pos, md, &b)
  $try_count += 1
  if re.respond_to? :to_str
  ...
```

要素技術

- グローバル変数
- 文字列の式展開
- 等差数列の和

グローバル変数

- $xxx のような変数はグローバル変数
- /¥A¥$[a-zA-Z_][a-zA-Z_0-9]*¥z/
- いままで使っていた pos とかはローカル変数
- グローバル変数はなるべく避ける

try の呼び出しを数える

```
def count_try(re, str)
  $try_count = 0
  rx_ends(re, str, 0)
  $try_count
end

def try(re, str, pos, md, &b)
  $try_count += 1
  if re.respond_to? :to_str
  ...
```

count_try の例

- count_try("a", "abc")　　#=> 1
  try("a") が呼ばれるだけ
- count_try([:cat, "a", "b"], "abc")　　#=> 3
  try([:cat, "a", "b"]) が
  try("a") と try("b") を呼ぶ

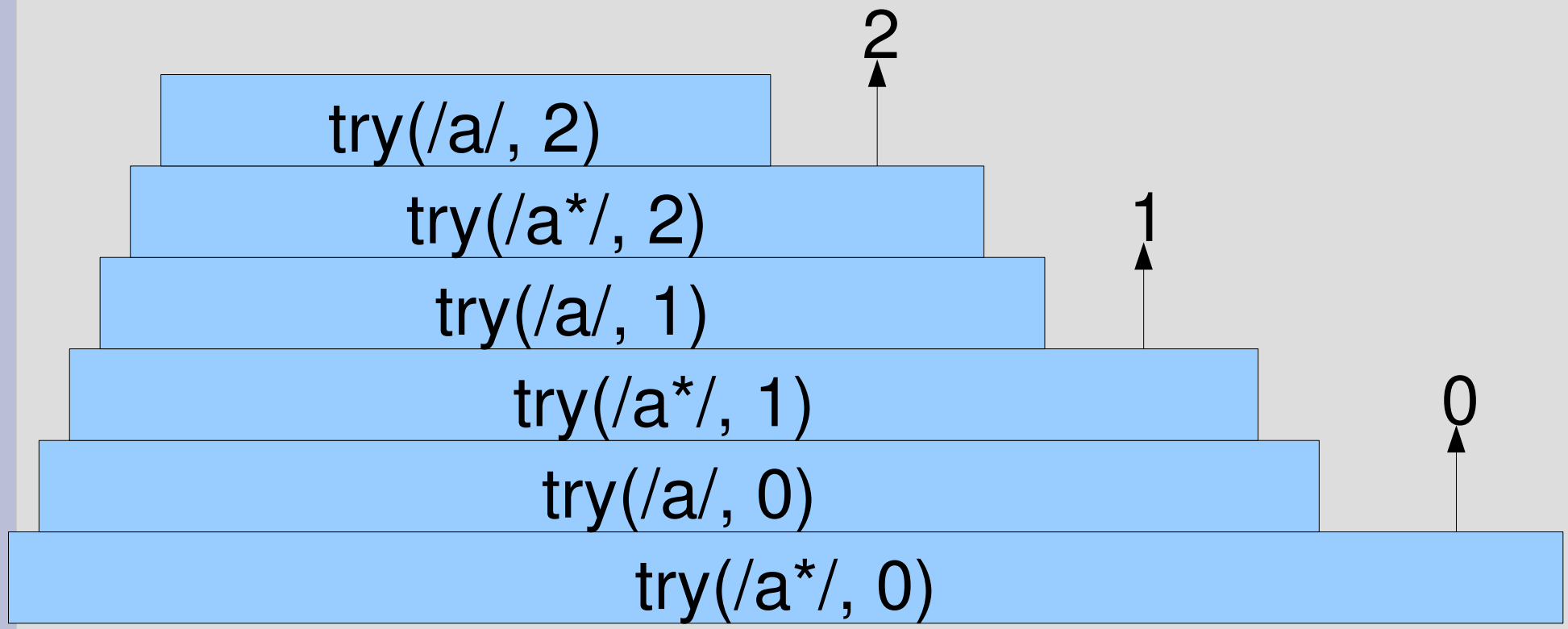
a* を a の並びにマッチするとき

- count_try([:rep, "a"], "aa") #=> 6
- count_try([:rep, "a"], "aaa") #=> 8
- count_try([:rep, "a"], "aaaaa") #=> 12
- count_try([:rep, "a"], "a"*10) #=> 22
- count_try([:rep, "a"], "a"*100) #=> 202
- count_try([:rep, "a"], "a"*1000) #=> 2002

- 文字列の長さを n として、2n+2 回呼び出される

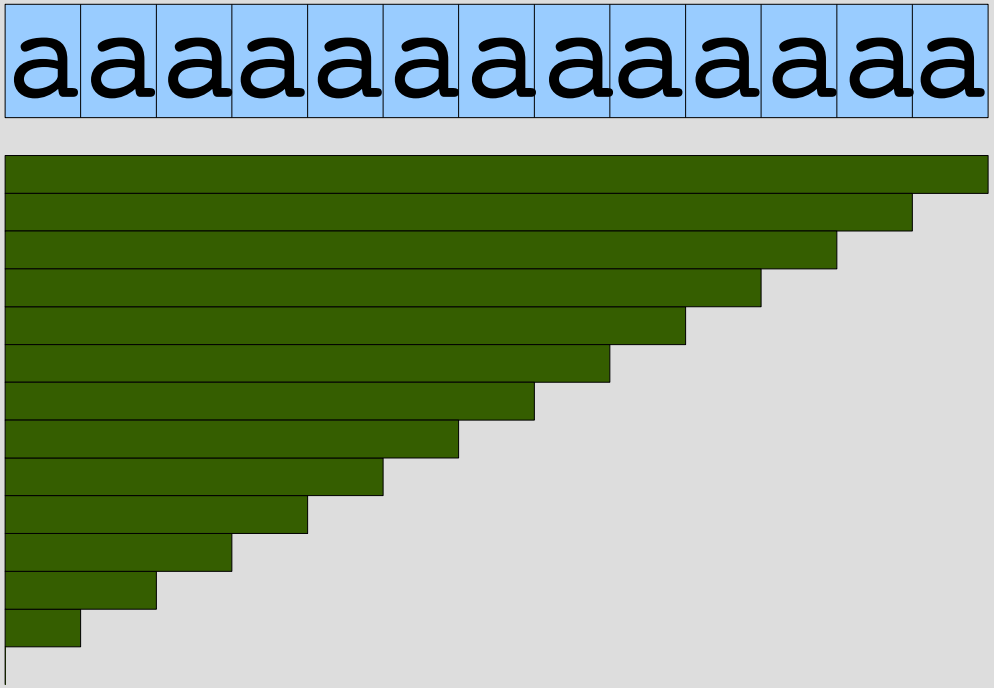
/a*/ =~ "aa"

- try(/a*/) が 3回
- try(/a/) が 3回
- 計6回



a*の処理

- a を可能な限り繰り返しマッチ
  - マッチしなくなったら繰り返しを止める
- 長い方から yield



aa に対するマッチ

- try([:rep, "a"])　　pos=0
  try("a")　　pos=0　マッチする
  try([:rep, "a"])　　pos=1
  try("a")　　pos=1　マッチする
  try([:rep, "a"])　　pos=2
  try("a")　　pos=2　マッチしない

- aa は長さ 2
- マッチする1文字ごとに 2回と、マッチしない最後の 2回で 2*2+2=6
- 一般に n文字なら 2n+2

0から20まで

```
0.upto(20) {|n|
  m = count_try([:rep, "a"], "a"*n)
  puts "#{n} #{m}"
}
=>    0 2
      1 4
      2 6
      3 8
      ...
```

文字列の式展開

- "aaa#{式}bbb" というように、文字列の中に式を埋め込める
- 埋め込んだ式は毎回評価されて結果が文字列として埋め込まれる
- ダブルクオートの文字列に使える
- シングルクオートの文字列には使えない

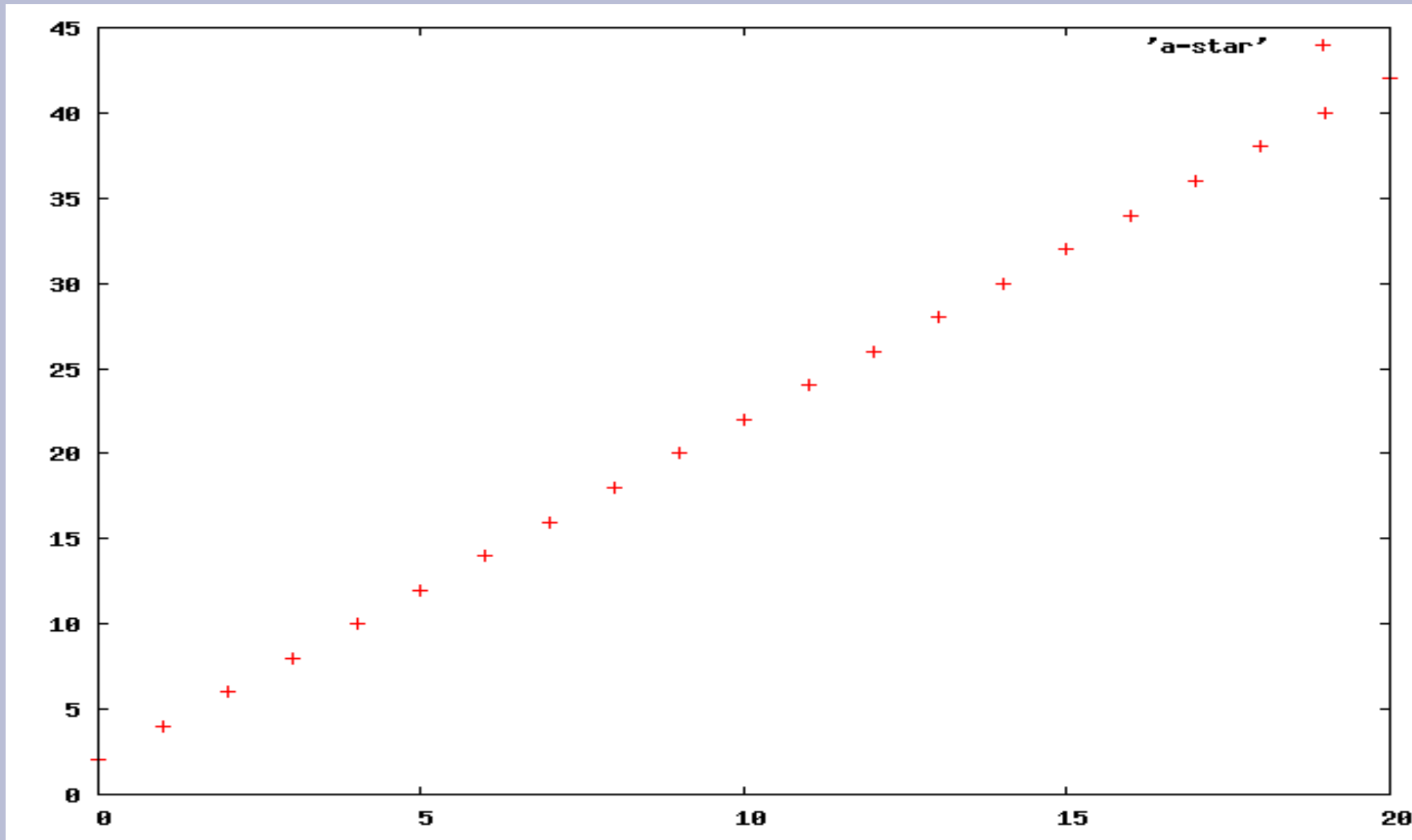
0から20まで

```
0.upto(20) {|n|
  m = count_try([:rep, "a"], "a"*n)
  puts "#{n} #{m}"
}
```

```
=>    0 2
      1 4
      2 6
      3 8
      ...
```

"#{n} #{count_try(...)}" と
直接埋め込んでもよい

0から20までのグラフ



a*a* を a の並びにマッチ

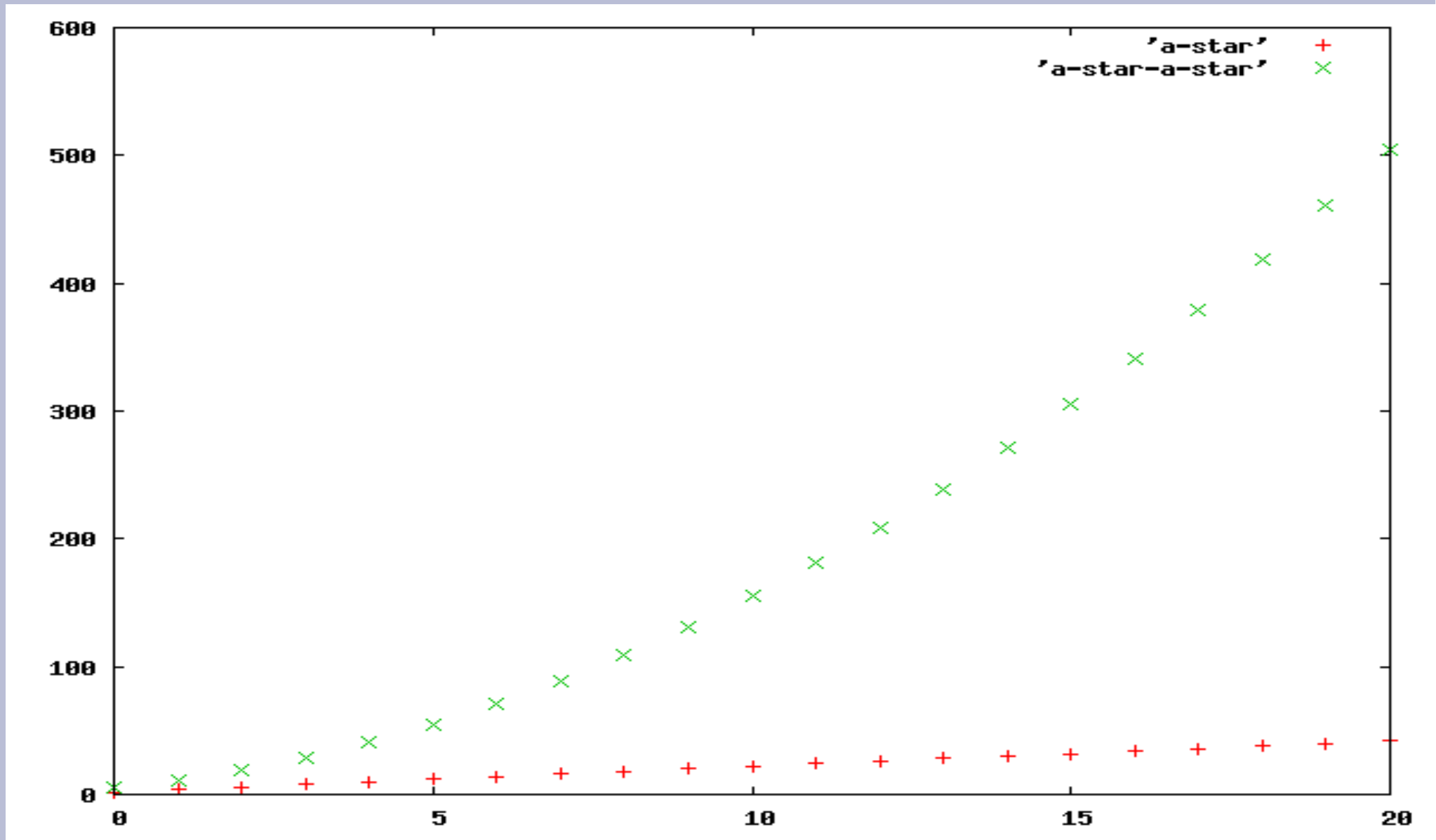
- a* と a*a* はどちらも a の並びにマッチする

```
0.upto(20) {|n|
  m = count_try([:cat, [:rep, "a"],
                       [:rep, "a"]], "a"*n)
  puts "#{n} #{m}"
}
```

```
=>   0 5    5 55   10 155  15 305  20 505
     1 11   6 71   11 181  16 341
     2 19   7 89   12 209  17 379
     3 29   8 109  13 271  18 419
     4 41   9 131  14 305  19 461
```

実は n^2+5n+5

a*a* と a のグラフ



a*a*の効率が悪い理由

- 最初の a* と後の a* の境目が曖昧だから曖昧ないろんな可能性すべてを検査するのは時間がかかる

aaに対するマッチ

- 最初の a* が aa にマッチする
  - 後の a* が空文字列にマッチする
- 最初の a* が a にマッチする
  - 後の a* が a にマッチする
  - 後の a* が 空文字列にマッチする
- 最初の a* が空文字列にマッチする
  - 後の a* が aa にマッチする
  - 後の a* が a にマッチする
  - 後の a* が空文字列にマッチする

a*a* を aa にマッチしたときの try

```
pos   exp
0     [:cat, [:rep, "a"], [:rep, "a"]]
0     [:rep, "a"]
0     "a"
1     [:rep, "a"]
1     "a"
2     [:rep, "a"]
2     "a"
2     [:rep, "a"]
2     "a"
1     [:rep, "a"]
1     "a"
2     [:rep, "a"]
2     "a"
0     [:rep, "a"]
0     "a"
1     [:rep, "a"]
1     "a"
2     [:rep, "a"]
2     "a"
```

最上位の try 呼び出し

最初の a* を伸ばせるだけ伸ばす

後の a* を位置 2 から伸ばす

後の a* を位置 1 から伸ばす

後の a* を位置 0 から伸ばす

a*a* を a の並びにマッチ

- 最上位の try 呼び出しで 1
- 最初の a* を伸ばすのに 2n+2
- ここまでで 2n+3

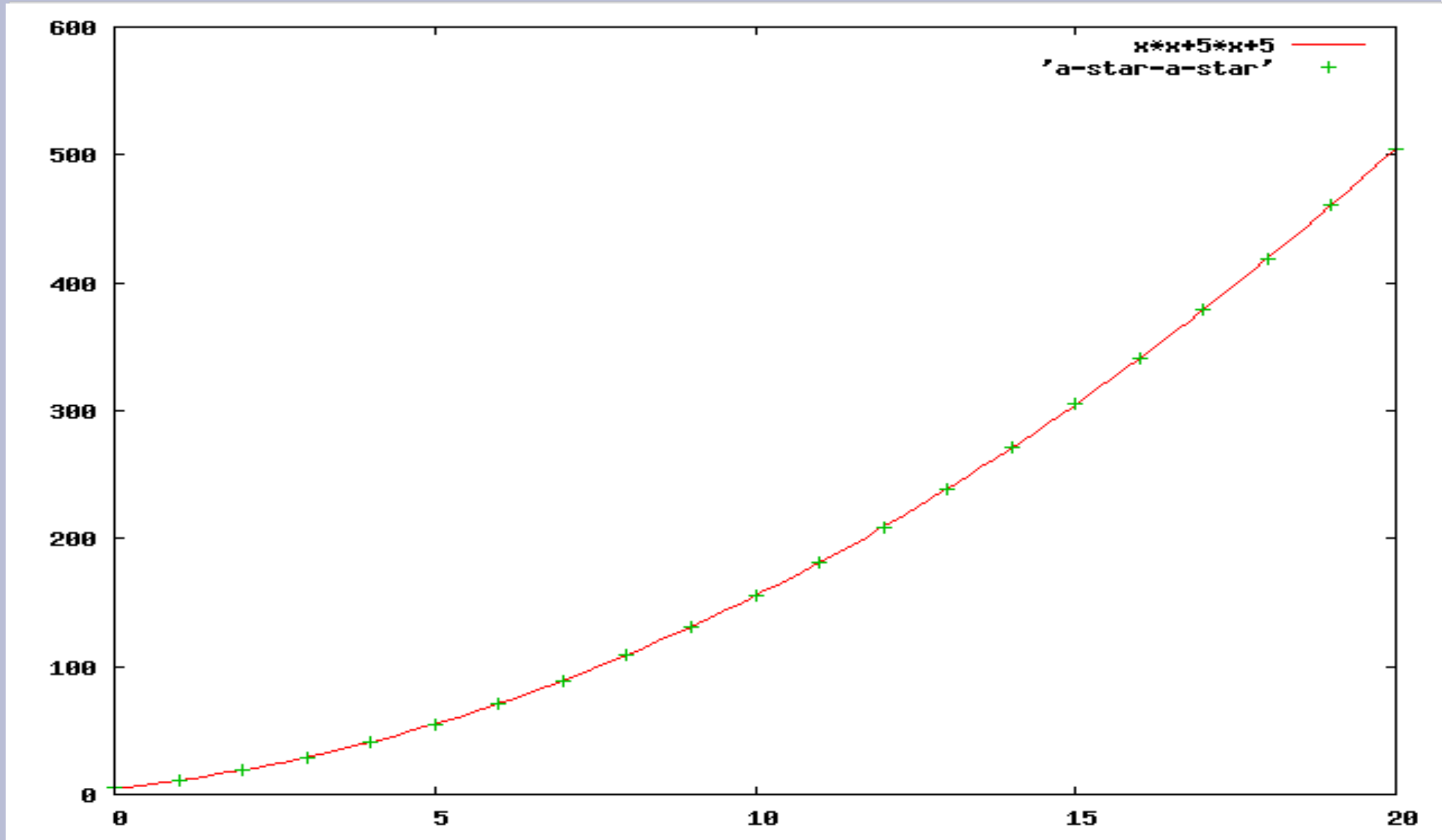
a*a* を a の並びにマッチ

- 前ページで 2n+3
- 後の a* で、2 + 4 + ... + (2n+2)=(n+1)(n+2)
  - 位置2から伸ばすのに 2
    - 残り長さ 0
  - 位置1から伸ばすのに 4
    - 残り長さ 1
  - 位置0から伸ばすのに 6
    - 残り長さ 2
- 計 $2n+3+(n+1)(n+2)=n^2+5n+5$

等差数列の和

- $1+2+\ldots+n = n(n+1)/2$

a*a*の実測値と理論値



(a*)* を a の並びにマッチ

- (a*)* も aの並びにマッチするのは a*、a*a* と同じ
- (a*)* は a*a* よりさらに効率が悪い

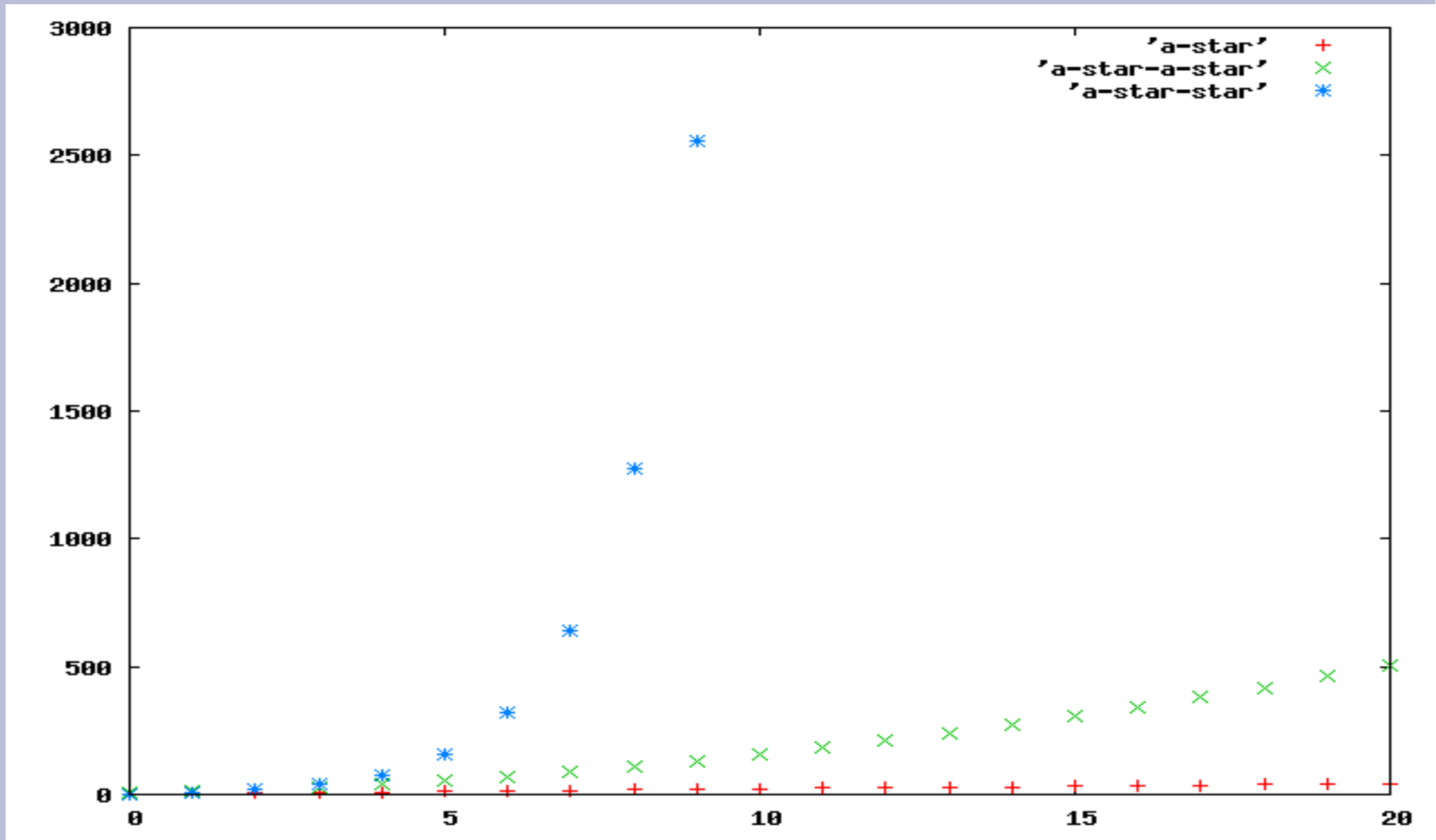
```
0.upto(20) {|n|
  m = count_try([:rep, [:rep, "a"]], "a"*n)
  puts "#{n} #{m}"
}
```

実は$5*2^n - 2$

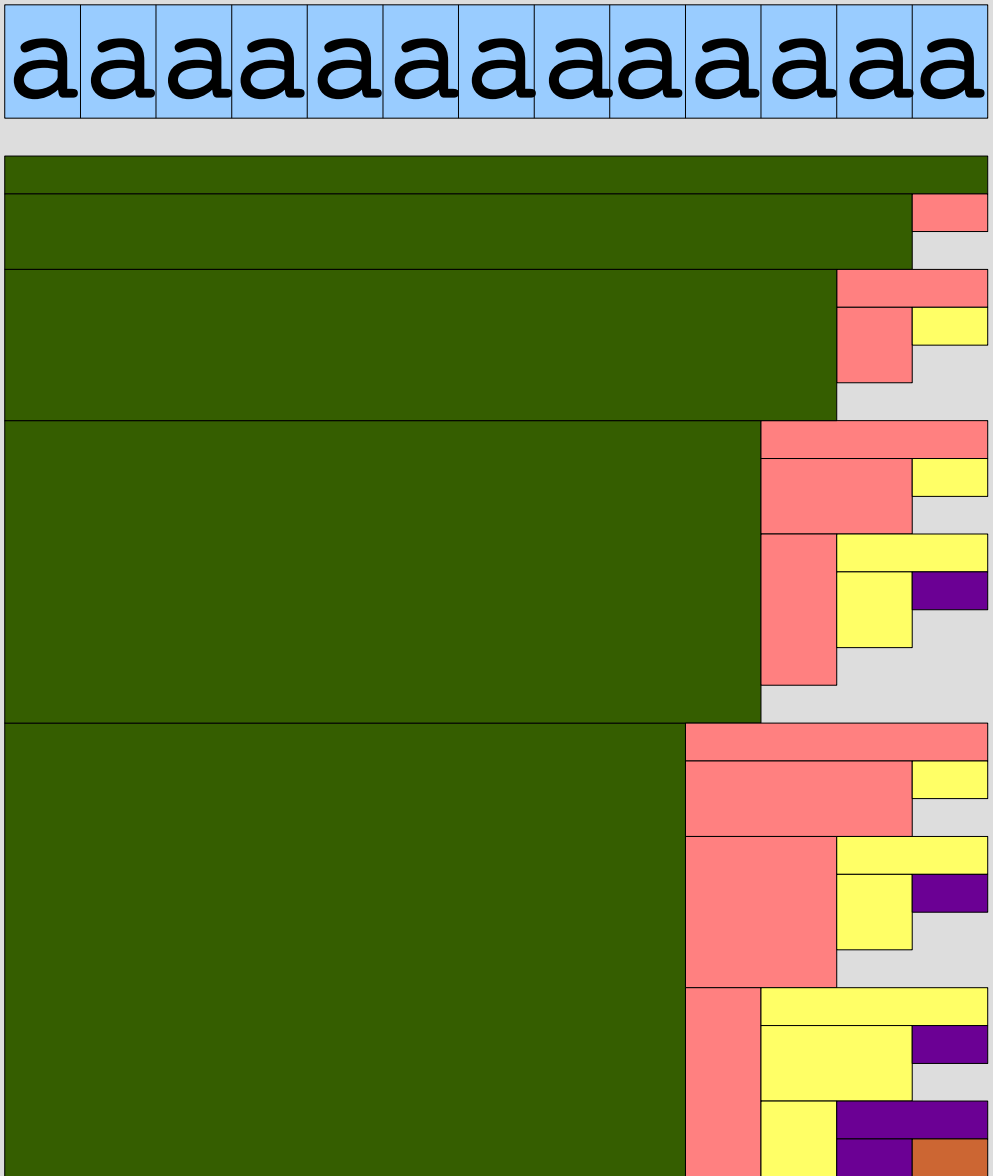
=>

0 3	4 78	8 1278	12 20478
1 8	5 158	9 2558	13 40958
2 18	6 318	10 5118	14 81918
3 38	7 638	11 10238	15 163838

(a*)*, a*a*, a* のグラフ


'a-star'
'a-star-a-star'
'a-star-star'
3000
2500
2000
1500
1000
500
0
0
5
10
15
20

(a*)*の効率が悪い理由



指数関数な実行時間

- count_try([:rep, [:rep, "a"]], "a"*20) が 1秒かかると仮定
  (手元の Pentium M 1.4GHz なマシンではもっとかかる)
- try 一回あたりの実行時間が一定と仮定
- $(5*2^{20}-2)*k = 1[sec] \quad => k = 1/5242878$

- 100文字の場合
  $(5*2^{100}-2)*k = 1.2 * 10^{24}$ [sec]

- 太陽の寿命 50億年 $= 5*10^{9}$ [年] $=1.6*10^{17}$[sec]

繰り返しの効率

- a* は 2n+2 回 try を呼び出す
- a*a* は n^2+ 5n + 5 回 try を呼び出す
- (a*)* は 5*2^n-2回 try を呼び出す
- aの並びという同じ対象にマッチするパターンでも、曖昧なものは遅くなる
- 繰り返しがネストしていると、とくに (指数関数的に) 遅い

レポート

- a が n 個並んでいる文字列に /a*aaaa/ をマッチさせたときに try が呼び出される回数を n に対する関数として求めよ
- try_cat, try_alt は任意個引数版を使う

- ∠切 2008-07-15 12:00
- RENANDI

/a*aaaa/ =~ "a" * n

- ある特定の n に対し、try の呼び出し回数は以下で求められる
- count_try(
  [:cat, [:rep, "a"], "a", "a", "a", "a"],
  "a" * n)

想定されるレポートの内容

- 求めた式 (関数)
- その式が求まった理由
  - グラフから求めるのではなく、プログラムの動作をたどって数える

ヒント

- わからなければ try の先頭に p exp とか p [pos, exp] とか入れて動作をたどる
- n が小さいときには場合分けが必要

まとめ

- 前回のレポートの解説
- 正規表現エンジンの停止性を説明した
- aの並びについて効率が悪いケースを述べた
- レポートを出した