プログラミング技法
第6回

教授　平木　敬

連絡先：hiraki@is.s.u-tokyo.ac.jp

本郷　理学部7号館409号室

今日のテーマは高速化

プログラム実行をどのように早くするか

高速化の技法

- 可能な技法は2つ
  1. 命令実行時間を短くする
     - IPCを多くする（Instructions per clock)
     - クロックを上げる
  2. 実行する命令数を少なくする
     - アルゴリズムを良くする
     - プログラム構造を良くする
     - コンパイラの最適化オプションを上げる
     - コンパイラを変える
     - ライブラリを変える

       ・・・・・・・・・・・・・・・・・

基本操作の速度

- 命令実行速度は一定でない
  - 10から100倍違うこともある
  - キャッシュメモリのヒット率
    - メインメモリはL1キャッシュの100倍くらい遅い
  - 分岐予測の成功率
    - 失敗すると分岐命令時間が10倍
  - 命令間の依存関係の有無
    - 依存関係が悪いと並列度が1になる（4倍遅い）
  - 演算器のとりあい

速度向上の工夫

- キャッシュメモリ利用による高速化
  - 速度差：L1 1、L2　3, L3　10, Main 100
    - 時間的局所性　　同じところを何度もアクセス
    - 空間的局所性　　近く（同じブロック内）を何度もアクセス
  - 一度キャッシュに入ったデータを使い尽くす
    - ブロッキング　データをブロック化して繰り返し使う
  - 命令間の依存関係の有無
    - 依存関係が悪いと並列度が1になる（4倍遅い）
  - 演算器のとりあい

コンパイラオプション等

- Cコンパイラ　様々な最適化オプション
  - 基本は　Oオプション
    - ０　Register宣言のないものは全部メモリにわりつけ
    - 1　出来るだけRegister割り付け、ループ構造などの簡略化、基本的な最適化
    - ２　関数inlining等以外の高度な最適化
    - ３　関数inliningを含む可能な限りの基本的最適化

Java　（実は速くならないという噂がある）

http://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html

Ruby　そういうものはない

ソフトウェアによる最適化

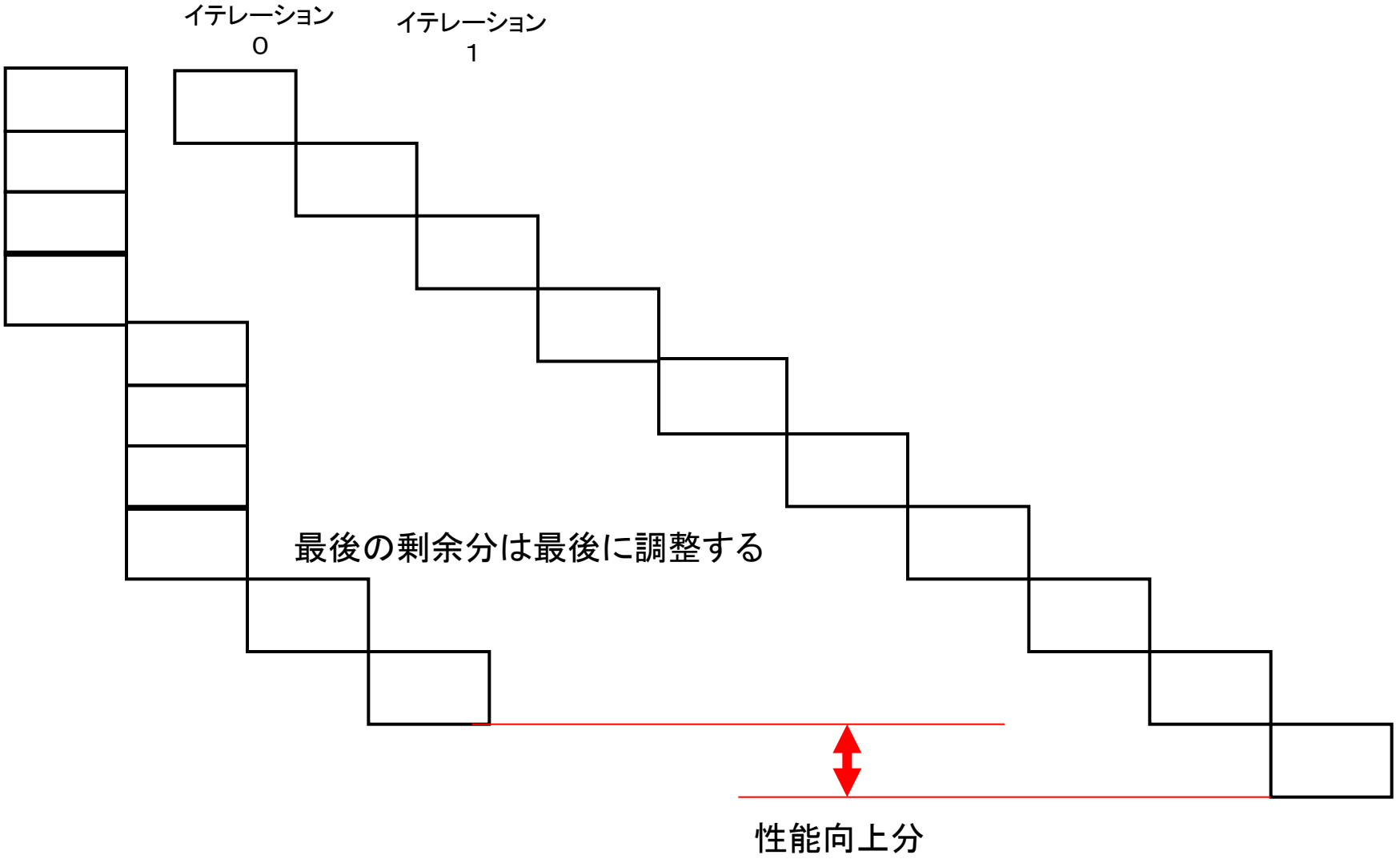
- ループアンローリング
  - ループを部分的に展開　⇒　多くのレジスタを使う
  - 条件分岐のオーバーヘッドを削減
  - アンロールした部分と、最後の微調整を分離

- ソフトウェア・パイプライニング
  - ループの展開に留まらず、展開し、命令をスケジューリングし、ループに再構成する
  - 異なるイテレーションに属する命令をループ内に混在

- トレース・スケジューリング
  - 基本ブロックに跨る命令群をスケジュールする
  - 高速化するトレースを決定し、それをスケジュール

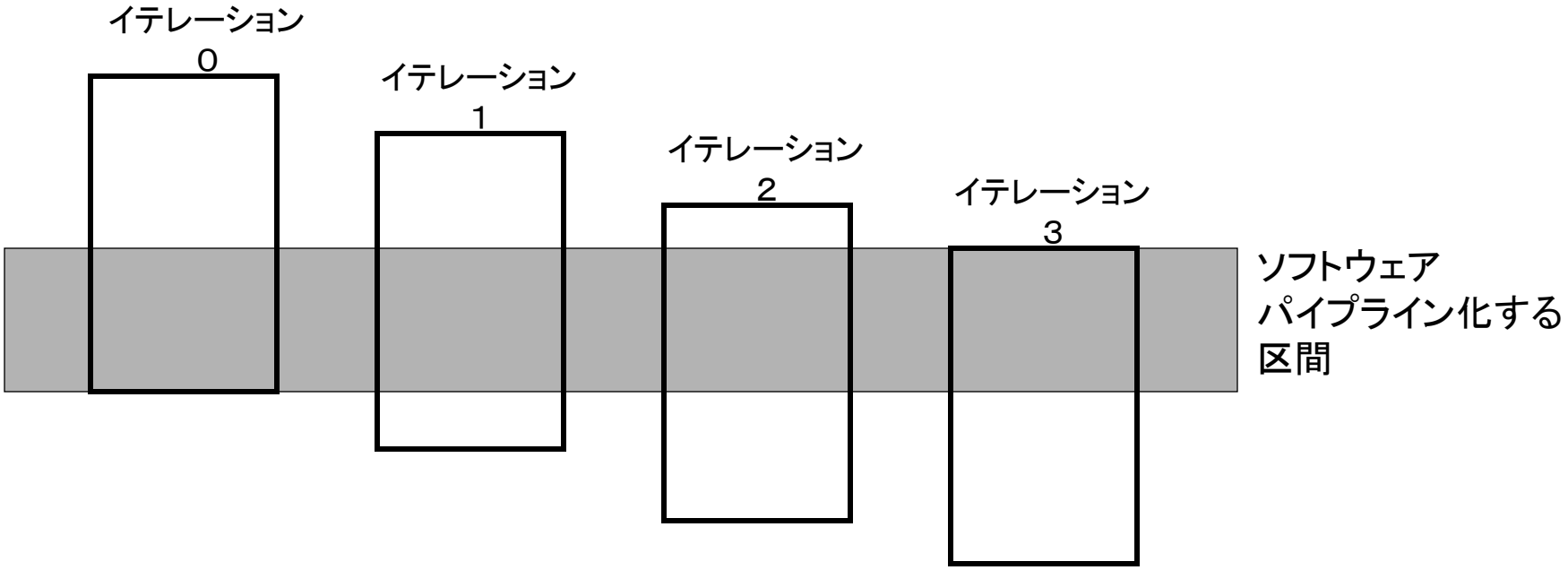
ループアンローリングの例

- ループをN回分まとめて展開する



ソフトウェア・パイプライニング

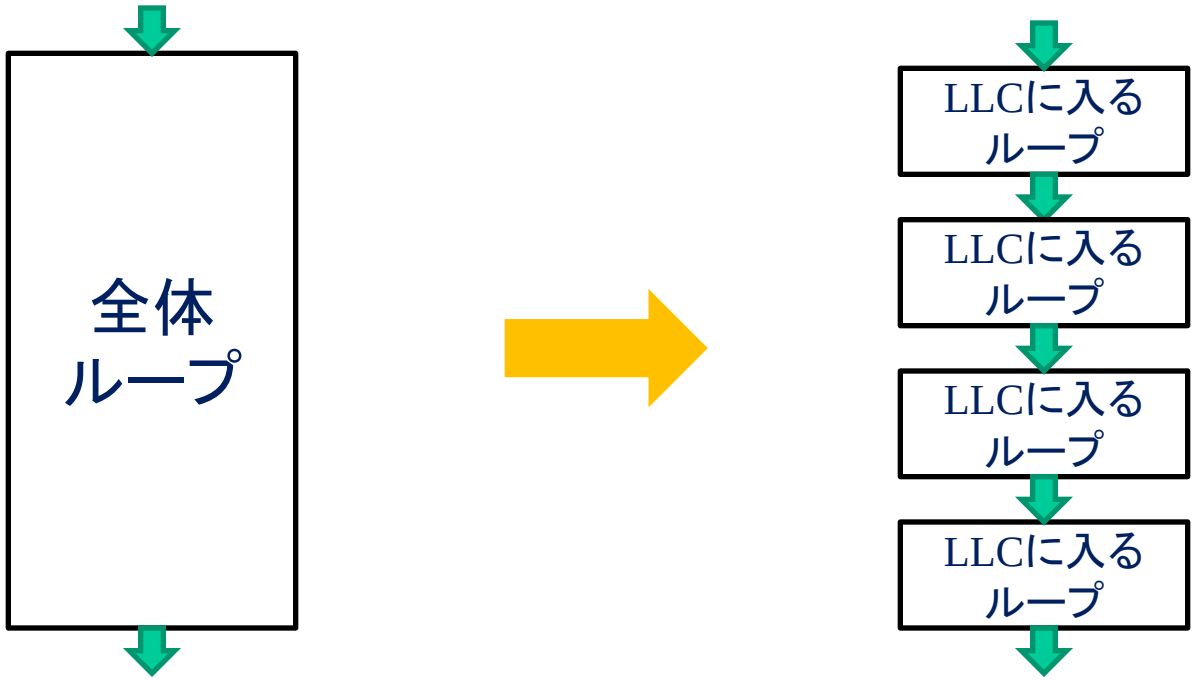
- シンボリックに、複数ループイテレーションを展開する



ソフトウェアパイプライン化により、ループ内命令のデータ依存距離を増大させる

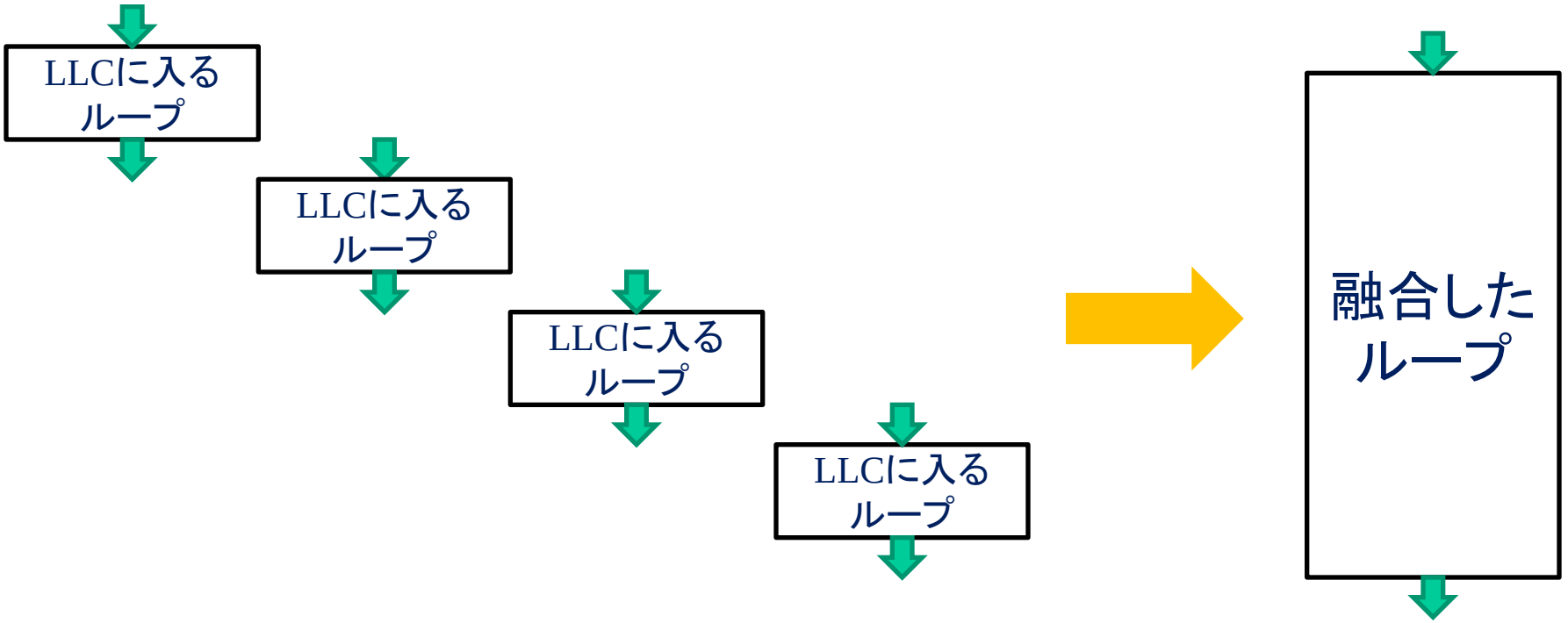
基本的ループ最適化

- ループ分裂 （Loop Fission)
  - データ構造全体のループを分割してキャッシュに入るようにする
    - キャッシュは多階層 ⇒ Last Level Cache （数MB）



基本的ループ最適化

- ループ融合 （Loop Fusion)
  - ループに分かれた操作をまとめて、オーバーヘッドを減らす
  - ループ分裂で分けたループをブロックごとにまとめる



最適化の実例

行列積のチューニング

行列積：基本プログラム

- A, B, C は $n \times n$ の行列とする
- $C := C + A \cdot B$ は以下のように書ける

```
for i := 1 to n
  for j := 1 to n
    t := C_ij
    for k := 1 to n
      t := t + A_ik * B_kj
    C_ij := t
```

行列積：基本プログラム

- A, B, C は $n \times n$ の行列とする
- $C := C + A \cdot B$ は以下のように書ける

```
for i := 1 to n
  for j := 1 to n
    t := C_ij
    for k := 1 to n
      t := t + A_ik * B_kj
    C_ij := t
```

外側ループ（outer loop）

中間ループ（middle loop）

内側ループ（inner loop）

性能の見積もり

- A, B, C は主記憶上に，i, j, k, n, t はレジスタ上にあると仮定すると

```
for i := 1 to n
  for j := 1 to n
    t := C_ij
    for k := 1 to n
      t := t + A_ik * B_kj
    C_ij := t
```

主記憶アクセス n^2 回

主記憶アクセス $2n^3$ 回，浮動小数点数演算 $2n^3$ 回

主記憶アクセス n^2 回

所要時間の大部分は内側ループにかかる

主に内側ループに着目する

性能の見積もり(続)

- 主記憶の帯域が 16 GByte/s とすると，倍精度浮動小数点数では 2GWord/s にあたる
- 浮動小数点数演算の帯域が 16 Gflops とすると

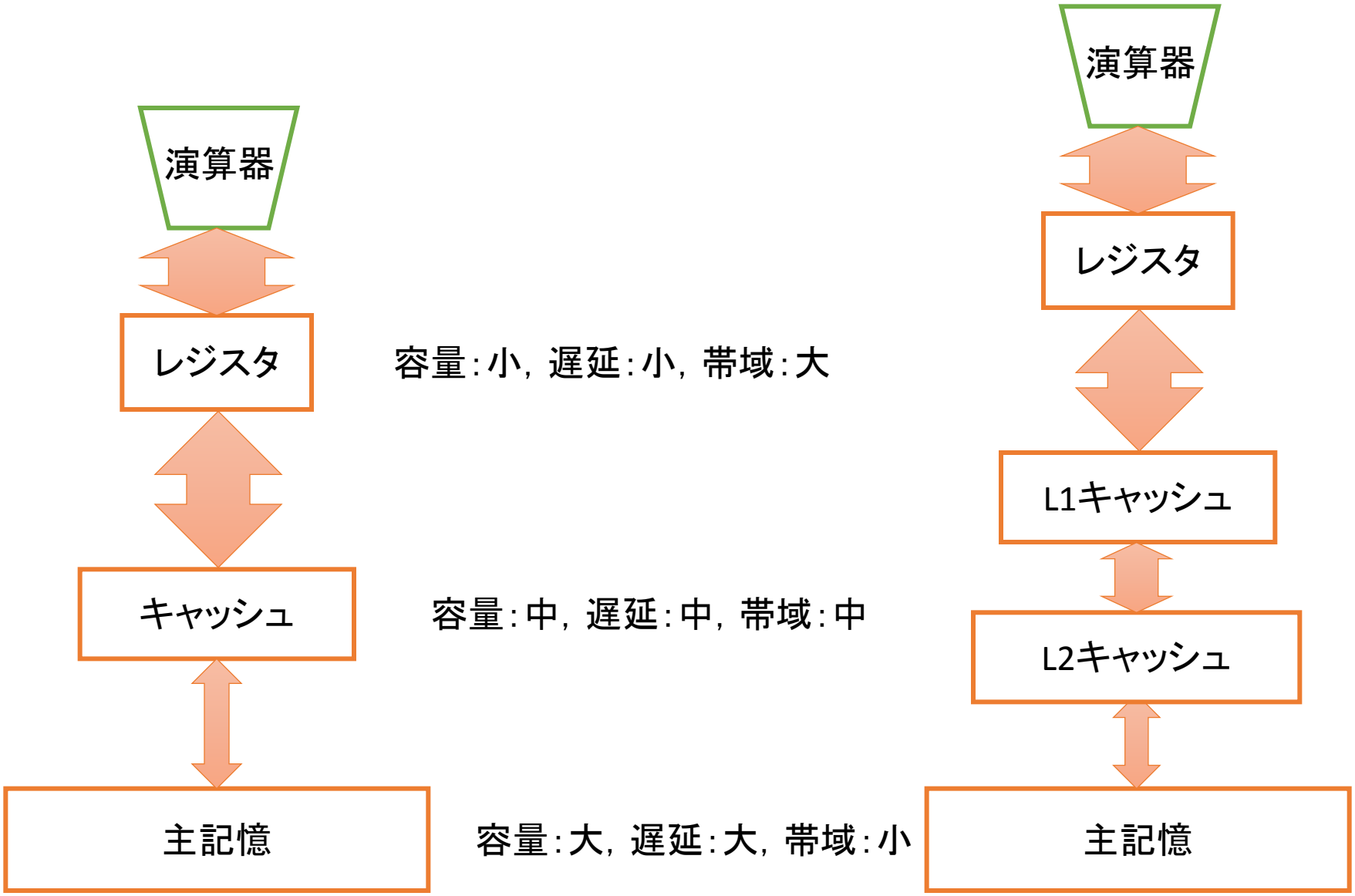
```
for i := 1 to n
  for j := 1 to n
    t := C_ij
    for k := 1 to n
      t := t + A_ik * B_kj
    C_ij := t
```

主記憶アクセス $2n^3$ 回，
浮動小数点数演算 $2n^3$ 回

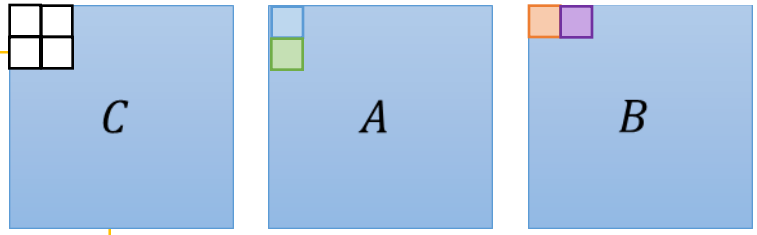
所要時間の 9 割近くは主記憶アクセスの時間になる

主記憶アクセスの数を減らす工夫をする

メモリ階層


演算器
レジスタ
キャッシュ
主記憶
容量：小，遅延：小，帯域：大
容量：中，遅延：中，帯域：中
容量：大，遅延：大，帯域：小
演算器
レジスタ
L1キャッシュ
L2キャッシュ
主記憶

最適化1：unrolling

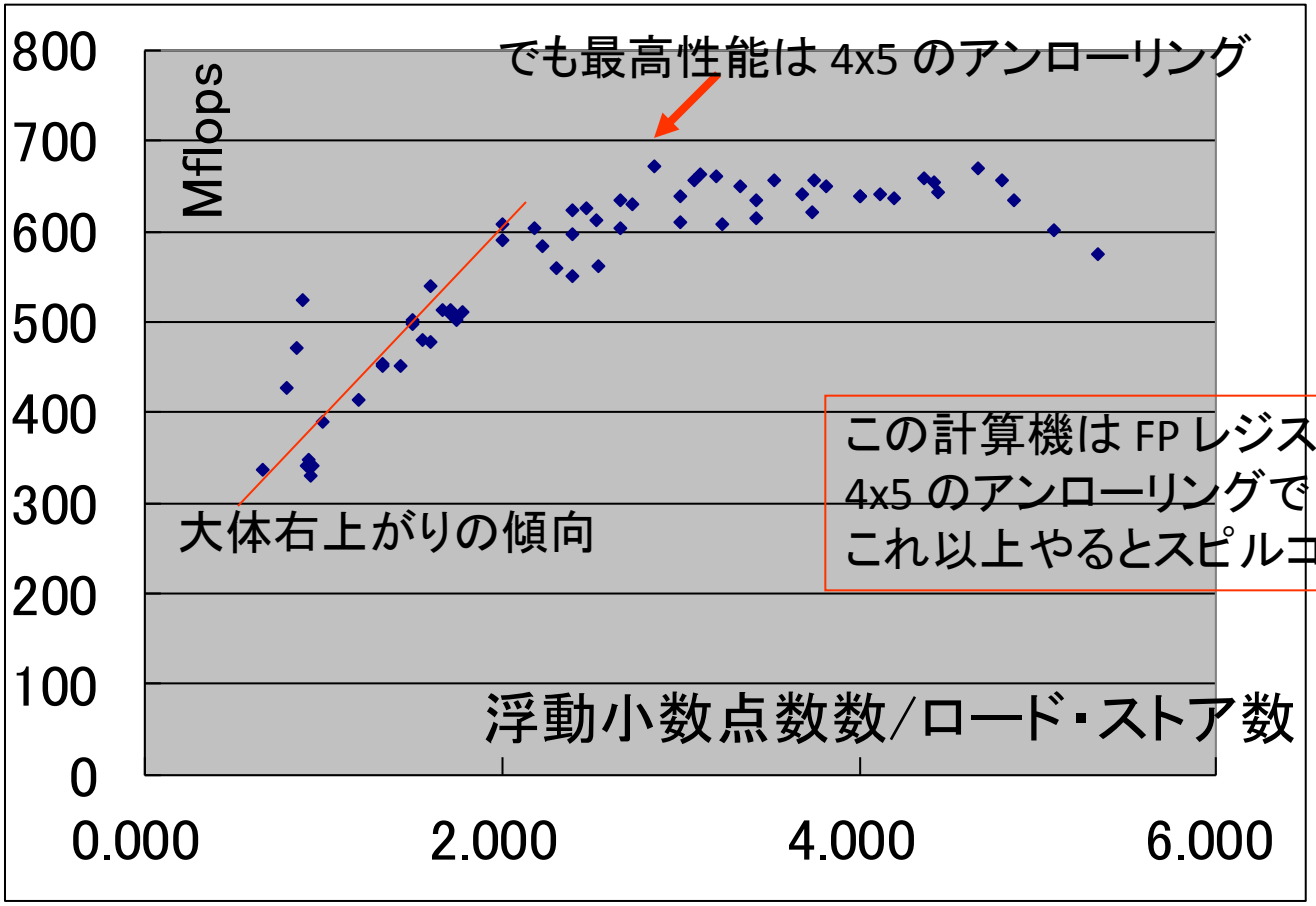


for $i := 1$ to n step 2
  for $j := 1$ to n step 2
    $t_{11} := C_{i,j}$; $t_{12} = C_{i,j+1}$
    $t_{21} := C_{i+1,j}$; $t_{22} := C_{i+1,j+1}$
    for $k := 1$ to n
      $t_{11} := t_{11} + A_{i,k} * B_{k,j}$
      $t_{12} := t_{12} + A_{i,k} * B_{k,j+1}$
      $t_{21} := t_{21} + A_{i+1,k} * B_{k,j}$
      $t_{22} := t_{22} + A_{i+1,k} * B_{k,j+1}$
    $C_{i,j} := t_{11}$; $C_{i,j+1} := t_{12}$
    $C_{i+1,j} := t_{21}$; $C_{i+1,j+1} := t_{22}$

2 回ずつ参照されるので，
メモリアクセス回数が ½ に
（レジスタを消費する）

※ 実際は n が奇数だった場合に対処するコードも必要

アンローリングと性能

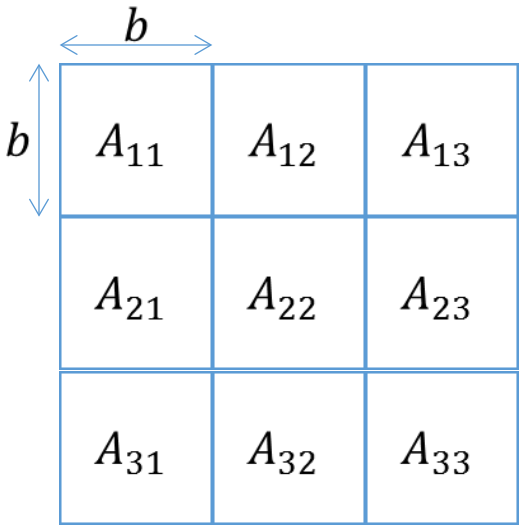

800
700
600
500
400
300
200
100
0
Mflops
でも最高性能は 4x5 のアンローリング
大体右上がりの傾向
この計算機は FP レジスタが 32 個
4x5 のアンローリングで 29 個必要
これ以上やるとスピルコードが出る
浮動小数点数数/ロード・ストア数
0.000
2.000
4.000
6.000

最適化2：blocking / tiling

- 各行列を小行列に分割する

```
for i_b := 1 to n step b
  for j_b := 1 to n step b
    for k_b := 1 to n step b
      for i := i_b to i_b + b − 1
        for j := j_b to j_b + b − 1
          t := C_ij
          for k := k_b to k_b + b − 1
            t := t + A_ik * B_kj
          C_ij := t
```

※ 実際は n mod b に対処するコードも必要

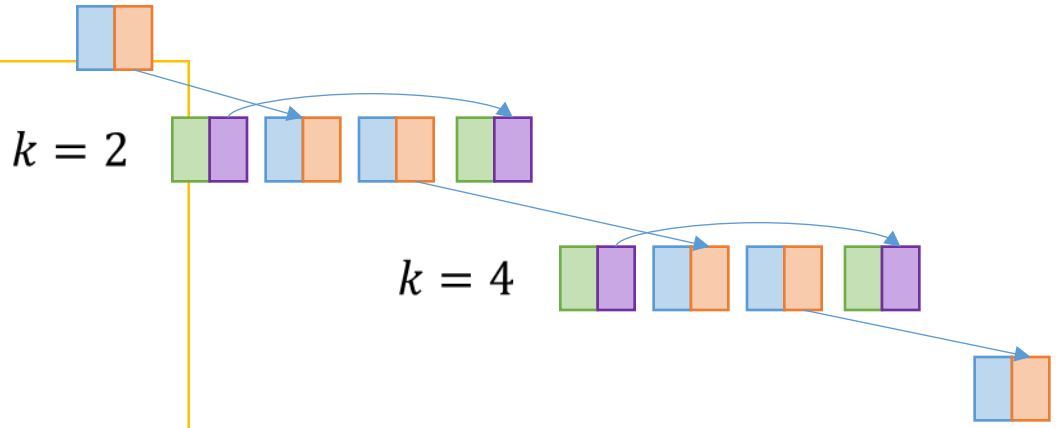


小行列どうしの積

キャッシュに小行列が3つ載れば，
主記憶アクセスは $1/b$ 倍にできる

最適化3：software pipelining

```
for i := 1 to n
  for j := 1 to n
    t := C_ij
    a_1 := A_i1; b_1 := B_1k
    for k := 2 to n step 2
      a_2 := A_ik; b_2 := B_kj
      t := t + a_1 * b_1
      a_1 := A_i+1,k; b_1 := B_k,j+1
      t := t + a_2 * b_2
    C_ij := t + a_1 * b_1
```



キャッシュアクセスの
遅延を隠蔽することができる

※ もっと深いパイプラインを組むことも多い

その他の最適化

- Prefetch
  - 後で使うデータを，あらかじめキャッシュに載せる
  - 特殊命令，またはコンパイラをうまくだます
- Copying
  - キャッシュ容量に合わせた一時配列を確保し，小行列をコピーする
  - Tiling だけではうまくいかないときに有効
- SIMD 化
  - CPU に内蔵した SIMD 演算器を利用する
  - コンパイラの特殊な指示，またはアセンブリ言語

最高性能を出すプログラム？

- 行列積のプログラムだけで多数の変種がある
- 最適な変種はハード・コンパイラに依存する

- PHIPACK
  - 行列積を計算する変種をシステマティックに生成する

- ATLAS
  - インストール時に性能を測り，もっとも高性能な変種を選んでインストールする（自動チューニングという）

実行命令数を減らす

実行命令数を減らす

- 非本質実行命令を削減する
  - OSのオーバーヘッドを減らす
    - 使わないプロセスを減らす（とても有効）
    - プログラムをプロセッサコアに固定する
    - GUIをCUIにする（とても有効）
    - 入出力をメモリ上に取る一時ファイルにする

実行命令数を減らす

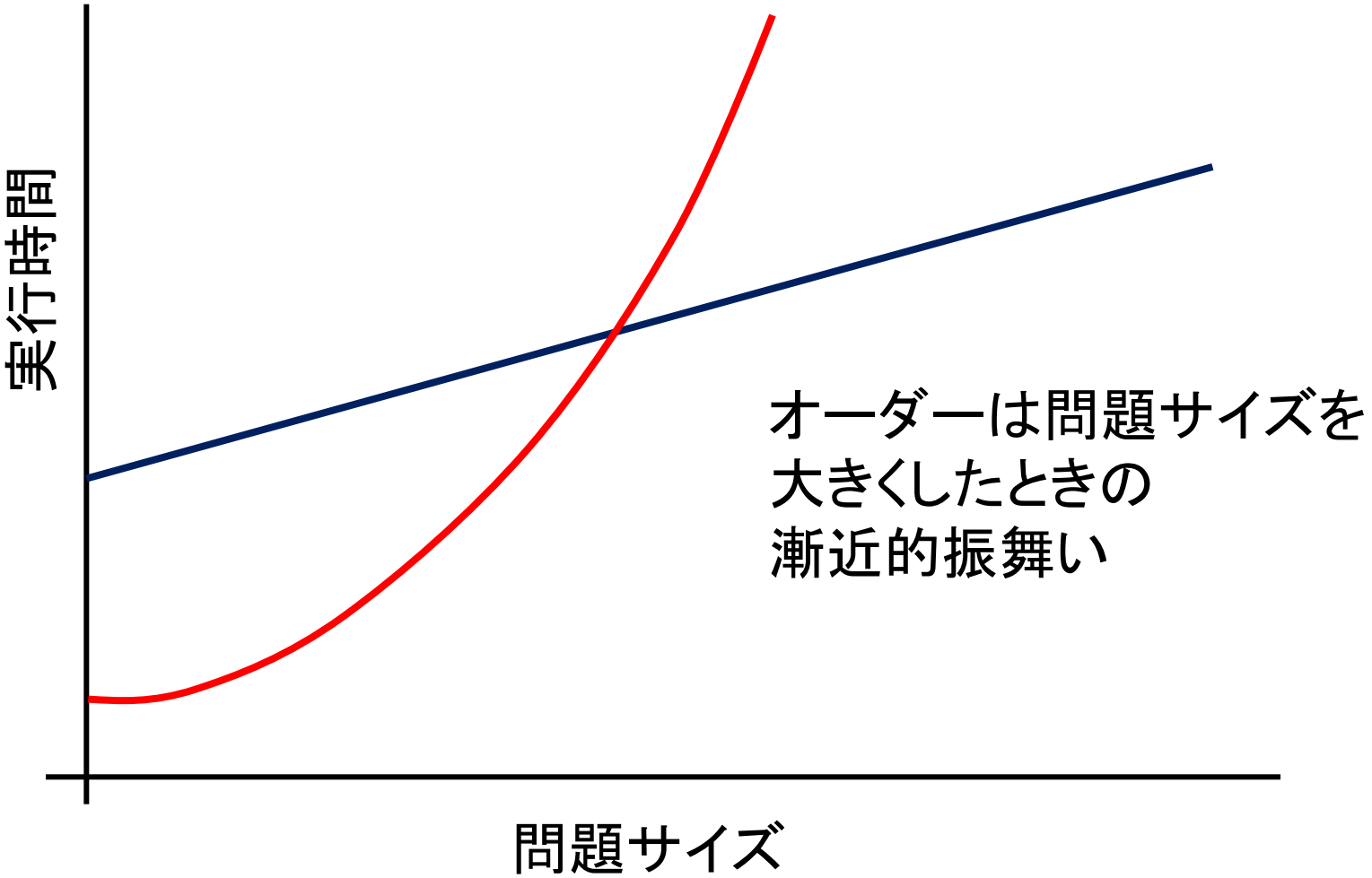
- プログラム中の無駄命令排除
  - 同じことの繰り返し計算は実は多い
    - 結果のRe-use　関数引数と内部状態でテーブル引き
    - 結果を投機　関数引数などで可能性の高い結果を予測
      正しい計算は別のコアで続け、予測結果で先に行く
    - 表引きを計算で置き換える
    - 計算を表引きで置き換える
    - 実行前の中間値実行をできるだけ行う

実行命令数を減らす

- アルゴリズムを変える
  - 全ての場合に最良なアルゴリズムは殆どない

実行時間とオーダー

- 例のプログラム　　バブルソート　　$O(n^2)$

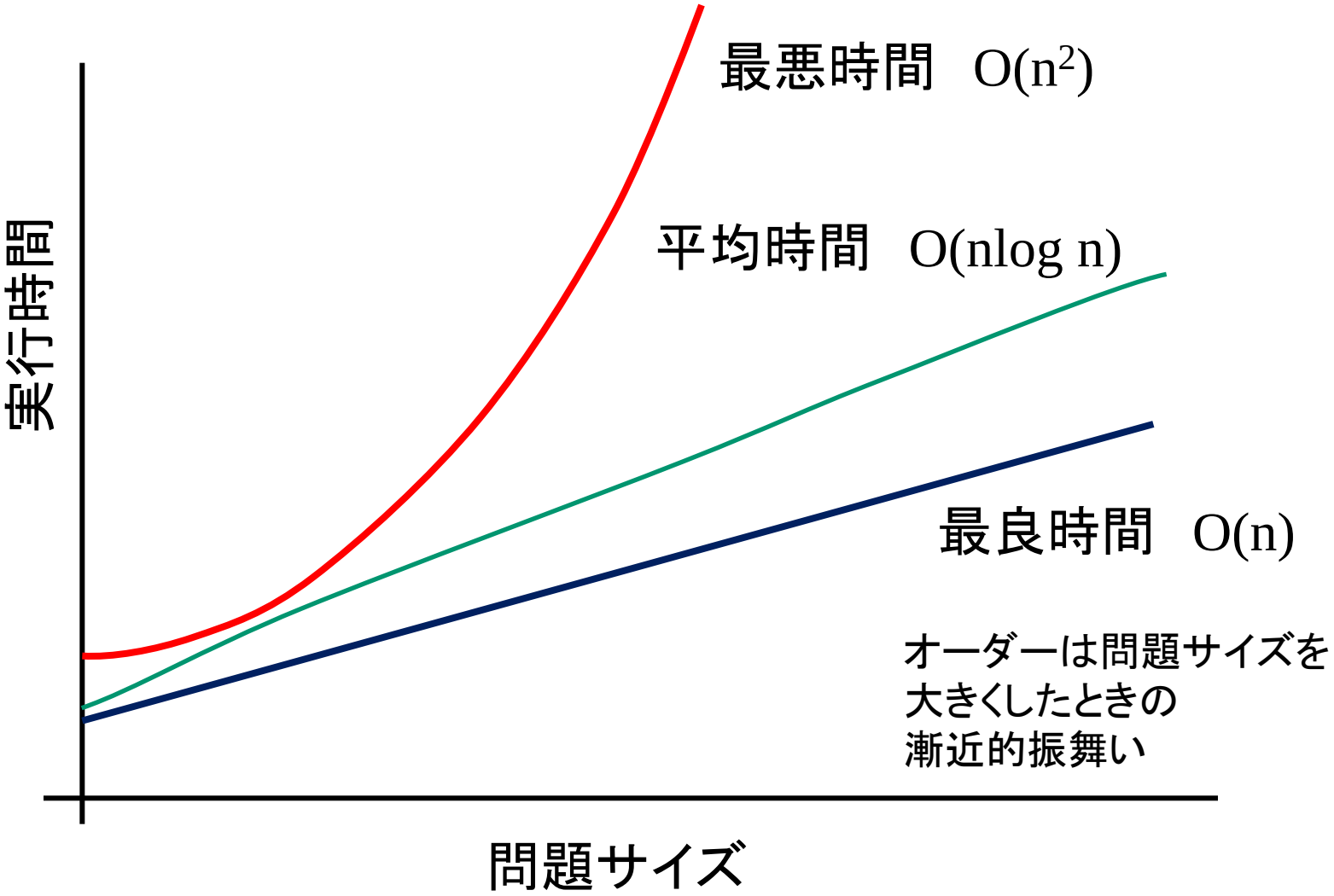


実行命令数を減らす

- アルゴリズムを変える
  - 全ての場合に最良なアルゴリズムは殆どない
- データ内容により速いとき、遅いときがある
  - 速い遅いが極端なもの、ばらつきが小さいものがある

最良・平均・最悪時間

- 例のプログラム　　クイックソート



アルゴリズムの最良・平均・最悪

- 上界、下界とも少し違う概念
- 平均は全ての入力パタンの結果の平均とか

- クイックソート　良く知られているが、実際にはさほど使われない
  - 最良　n に比例(最初から整列している時）
  - 平均　n log n
  - 最悪　n^2　に比例（入力データが逆順に整列）

入力に癖があることは、場面によるが少なくない

実行命令数を減らす

- アルゴリズムを変える
  - 全ての場合に最良なアルゴリズムは殆どない
- データ内容により速いとき、遅いときがある
  - 速い遅いが極端なもの、ばらつきが小さいものがある
- アルゴリズムを切り替えながら使う
  - Hybrid　サイズやランダムサンプリングで切替