Binary Indexed Tree のはなし

保坂 和宏 (東京大学理学部数学科)

第 13 回 JOI 春合宿

2014/03/19

概要

- Binary Indexed Tree とは
  - 何ができる？
  - 何が嬉しい？
- 具体的な実装
- 応用範囲
  - 区間に足す問題
  - 多次元
  - 二分探索

目標

- 実装できるようにする
- 「普通に Binary Indexed Tree を使うだけ」の部分で詰まらないようになる
  - 補助的な道具としてぱっと使えるように

Binary Indexed Tree とは

Binary Indexed Tree

- Binary Indexed Tree (Fenwick Tree)
  - Peter M. Fenwick, "A New Data Structure for Cumulative Frequency Tables" (1994)
  - BIT と呼ぶことにします
- 列に対するある種の処理ができる

基本的な問題

- N 個の変数 $v_1, \ldots, v_N$
  - すべて 0 で初期化
- 2 種類のクエリ
  - v_a に値 w を加える
  - prefix $[1, a]$ のところの和 $v_1 + v_2 + \cdots + v_a$ を求める
- クエリあたり $O(\log N)$ 時間にしたい

それ○○でもできるよ！

- それ平衡二分探索木でもできるよ！
  - `std::set` 等では機能が足りず，自分で木を実装することに
  - 実装量と計算量がたくさん倍になります
- それ Segment Tree でもできるよ！
  - 実装量と計算量が数倍になります
  - が，割と現実的な選択肢
    - (少なくともコンテストにおいては)

それ○○でもできるよ！

- 詳しくは，例えば秋葉さんの講義を参考に
  - 平衡二分探索木
    - プログラミングコンテストでのデータ構造 2 (2012)
      - http://www.slideshare.net/iwiwi/2-12188757
  - Segment Tree
    - プログラミングコンテストでのデータ構造 (2010)
      - http://www.slideshare.net/iwiwi/ss-3578491

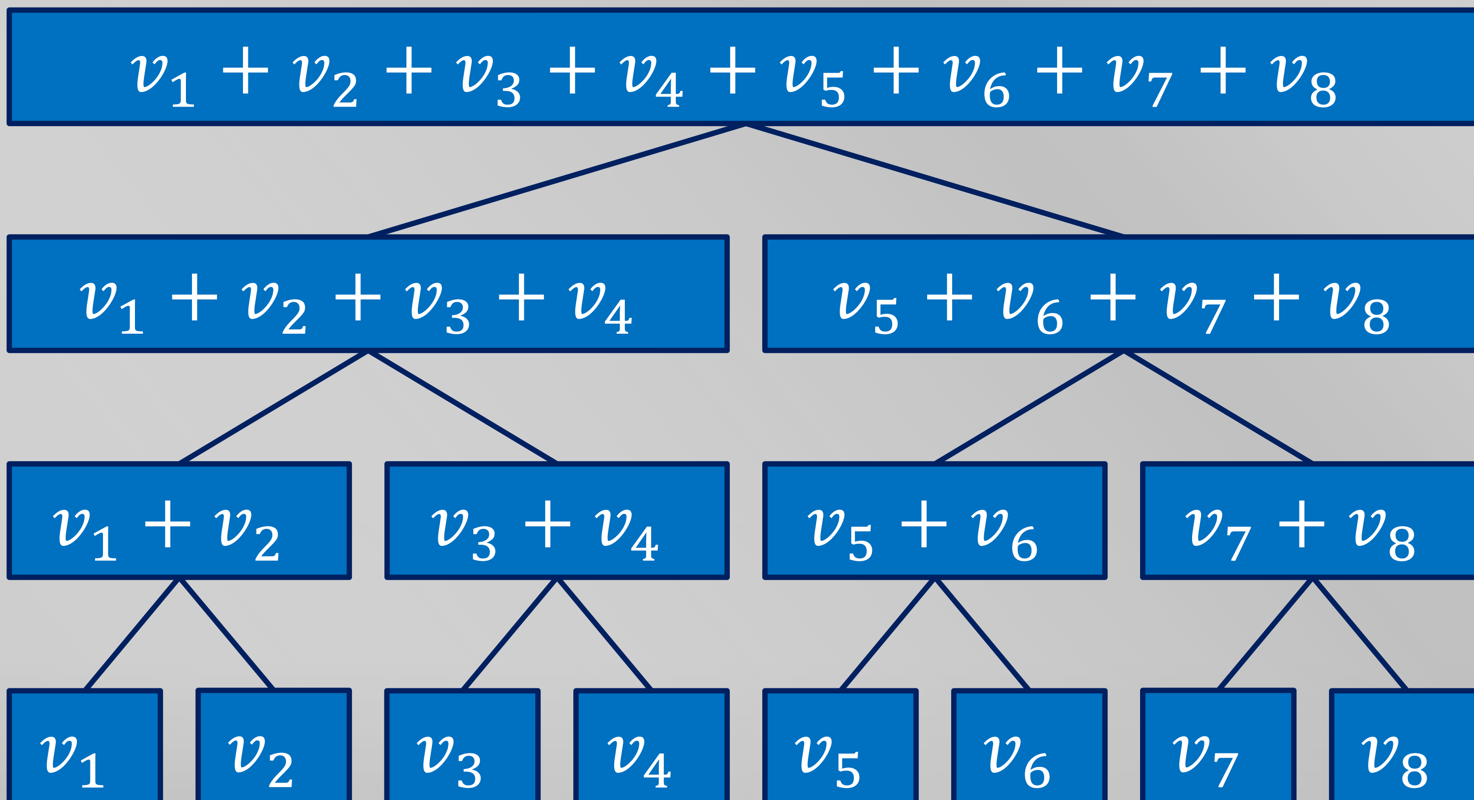
Binary Indexed Tree の特徴

- サイズ N の配列で実現できる！
- 速い！
- 実装が簡単！
- 応用範囲はあまり広くない
  - Segment Tree の機能を制限して単純化したものと考えられる

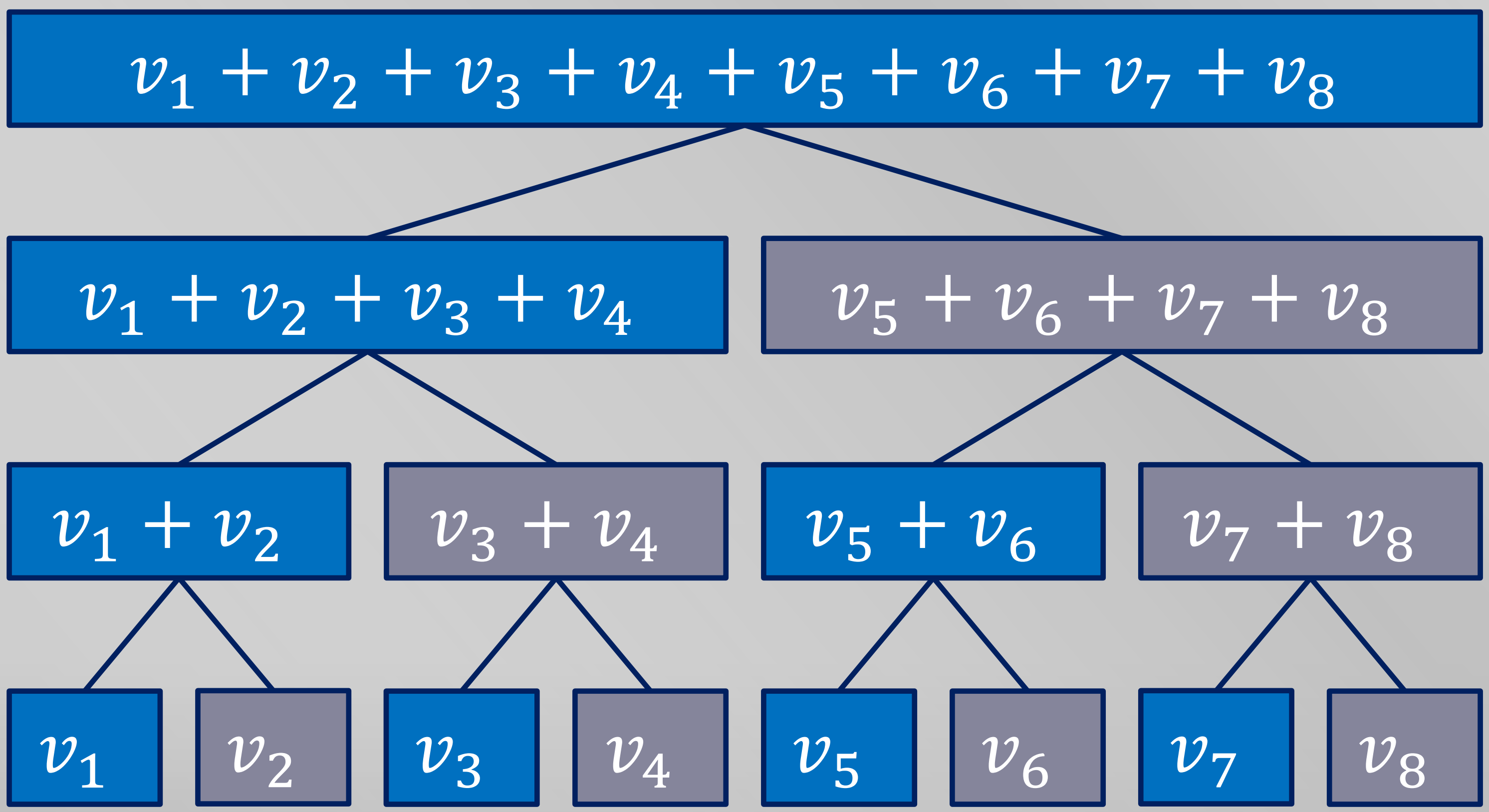
基本的な問題

- N 個の変数 $v_1, \ldots, v_N$
  - すべて 0 で初期化
- 2 種類のクエリ
  - v_a に値 w を加える
  - prefix $[1, a]$ のところの和 $v_1 + v_2 + \cdots + v_a$ を求める
- クエリあたり $O(\log N)$ 時間にしたい

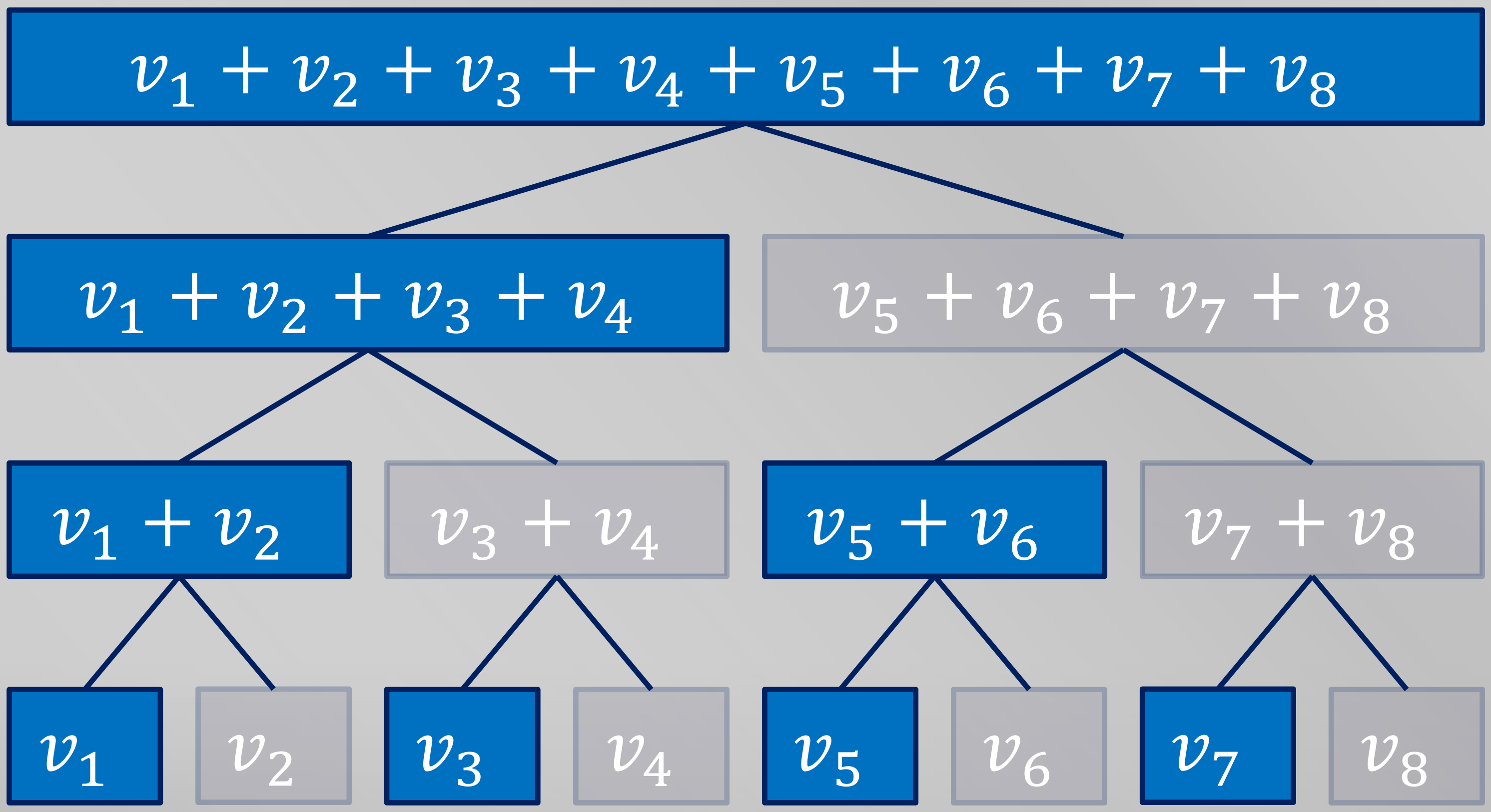
Segment Tree のアイデア



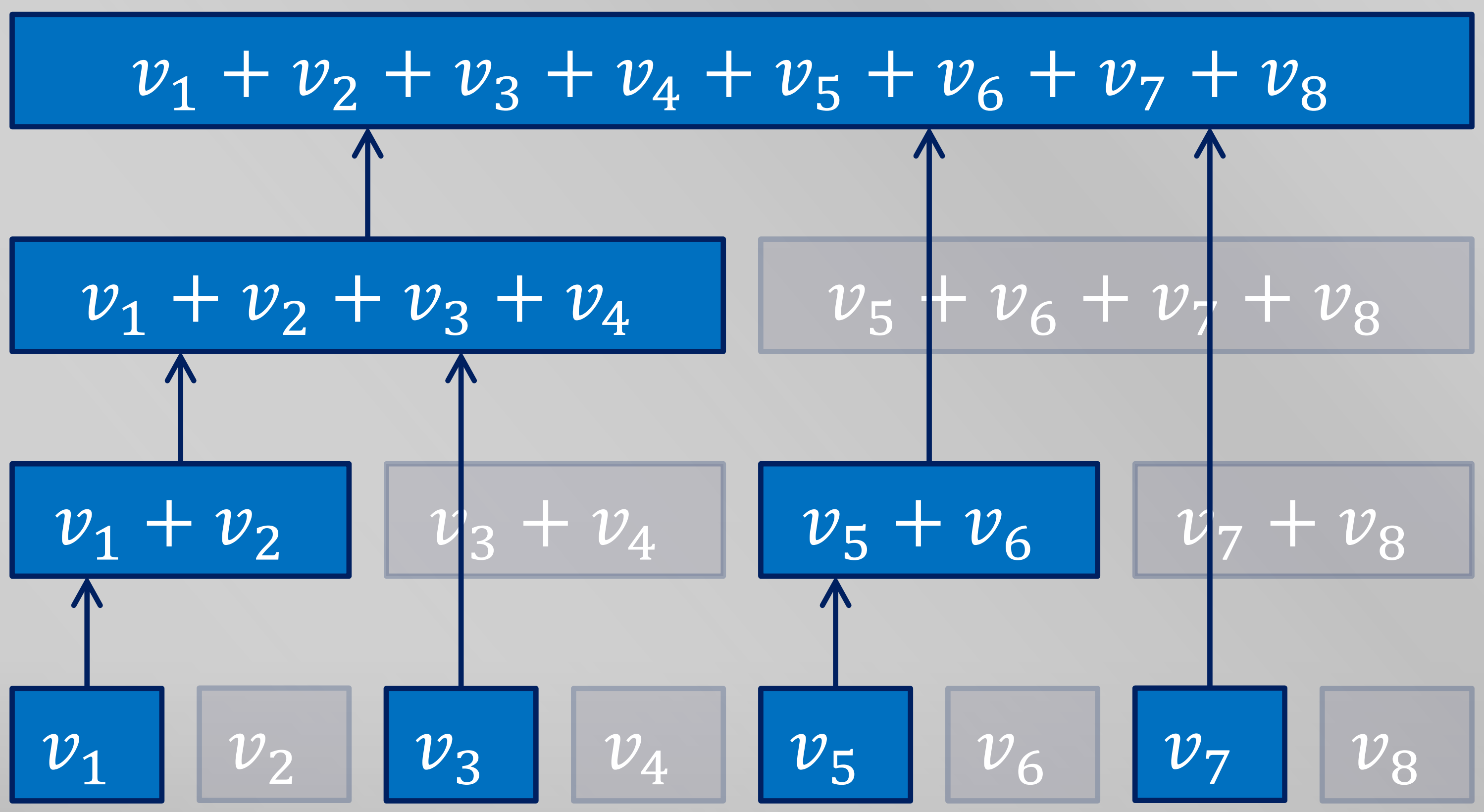
Binary Indexed Tree のアイデア



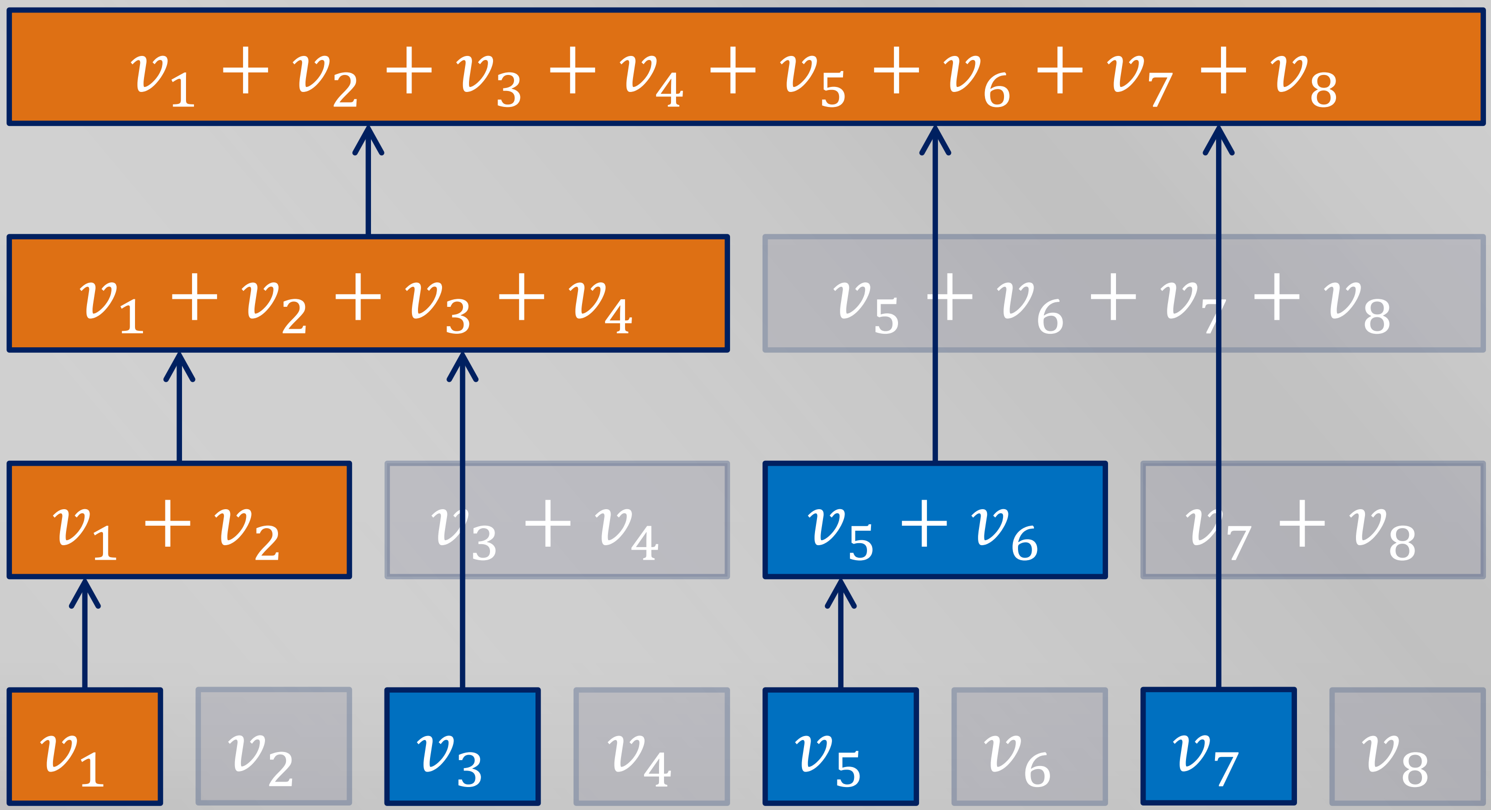
Binary Indexed Tree のアイデア



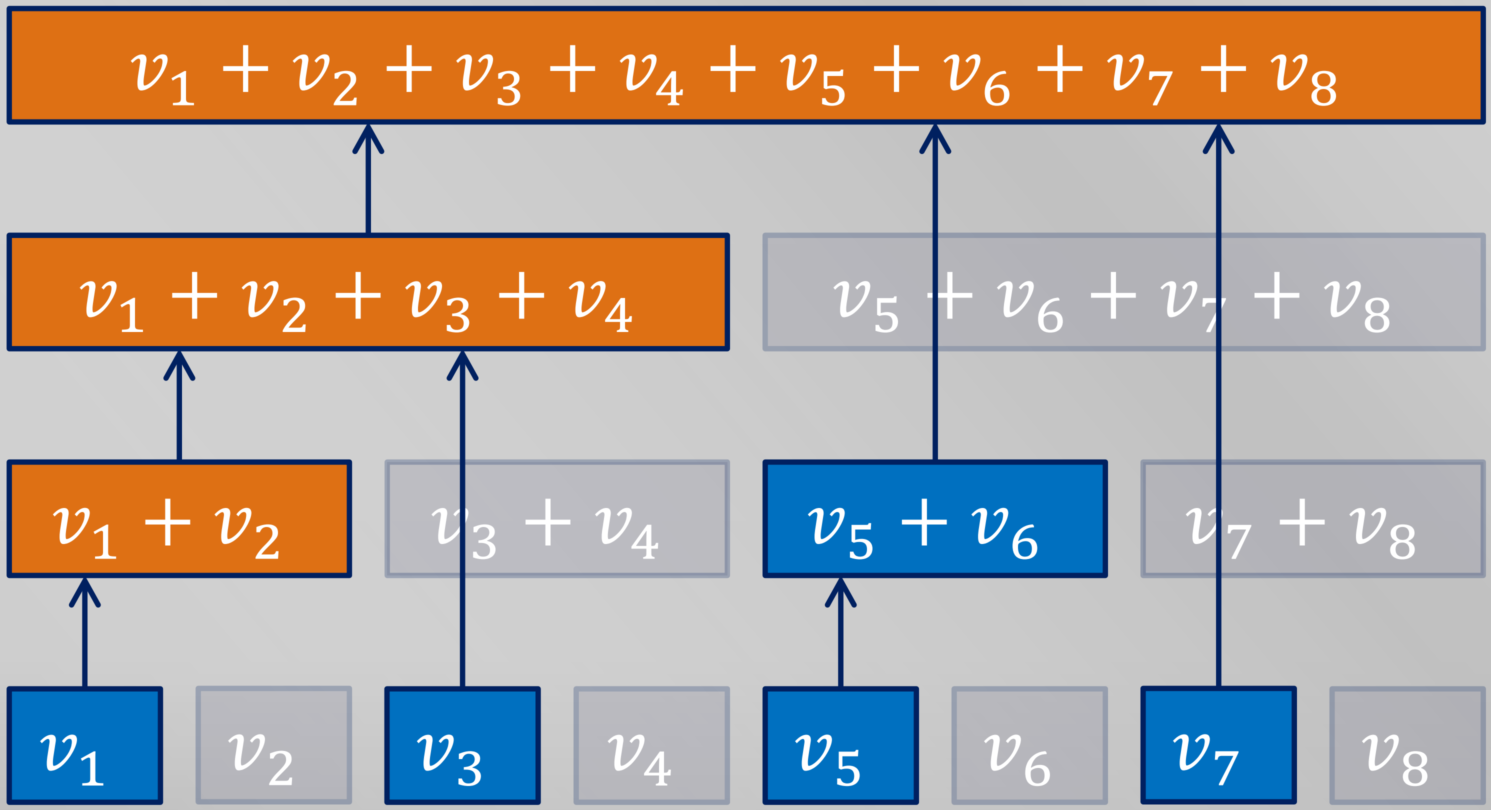
変数の値の更新


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

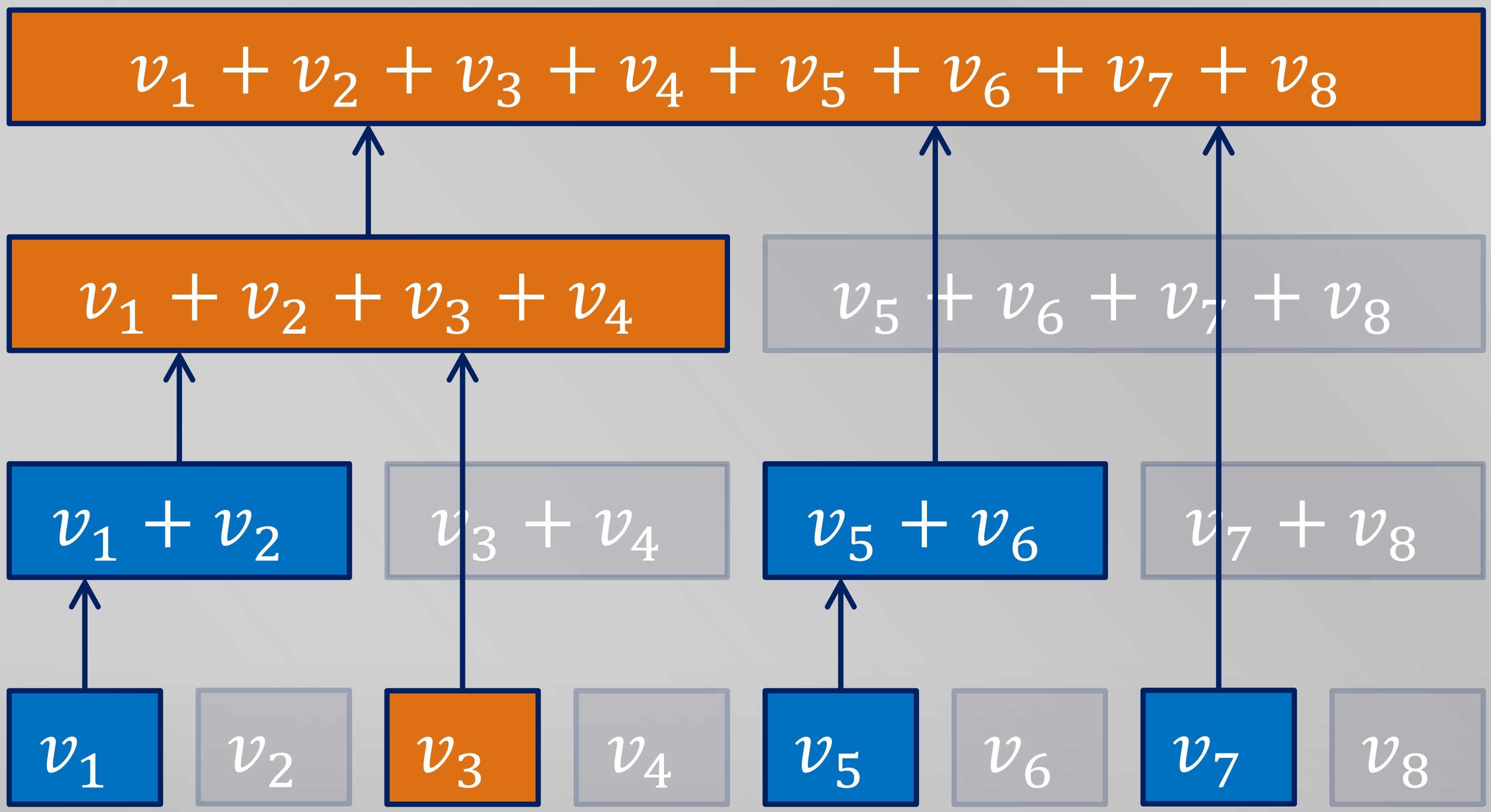
変数の値の更新 (v_1)


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

変数の値の更新 (v_2)

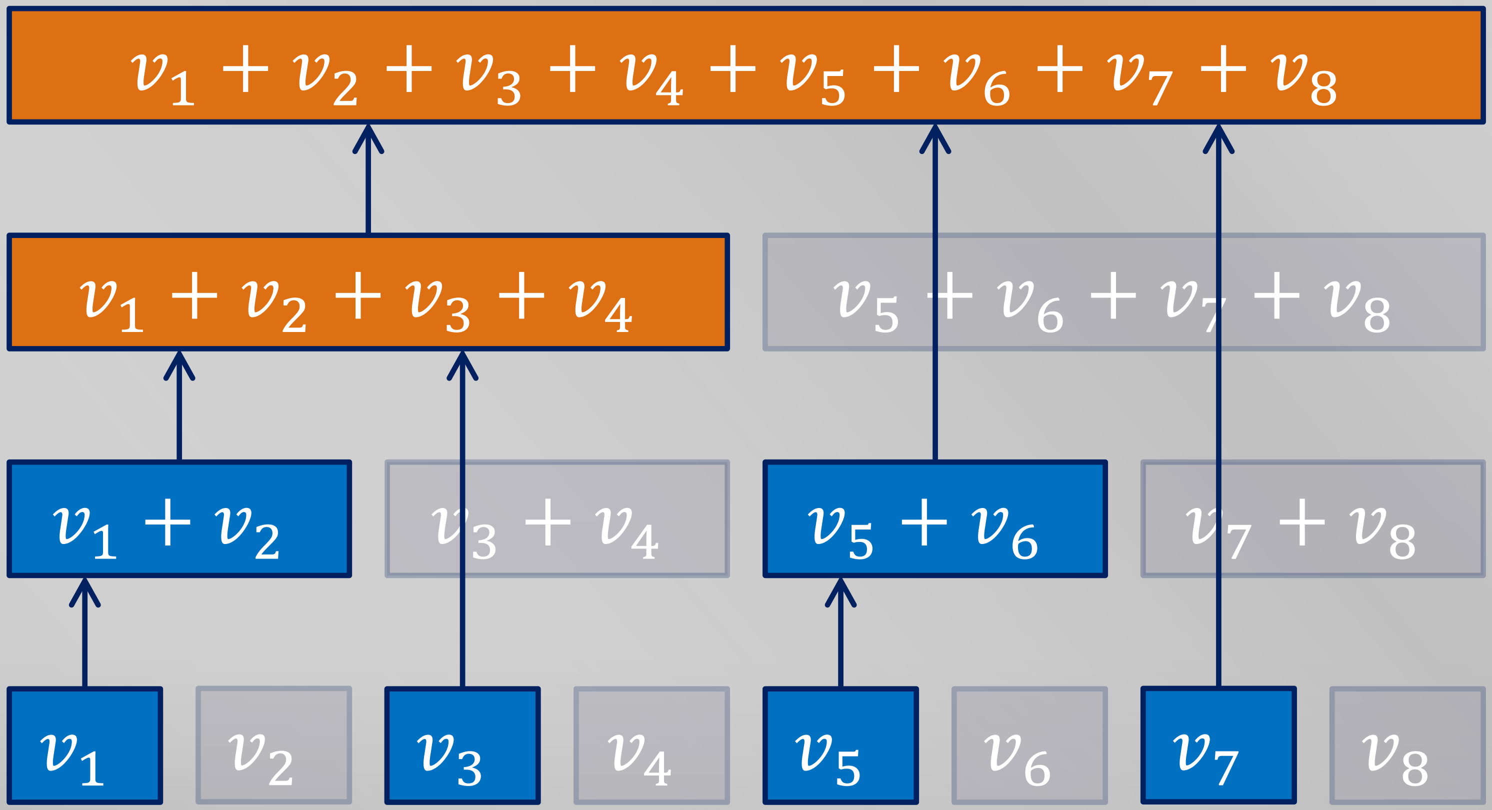

$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

変数の値の更新 (v_3)

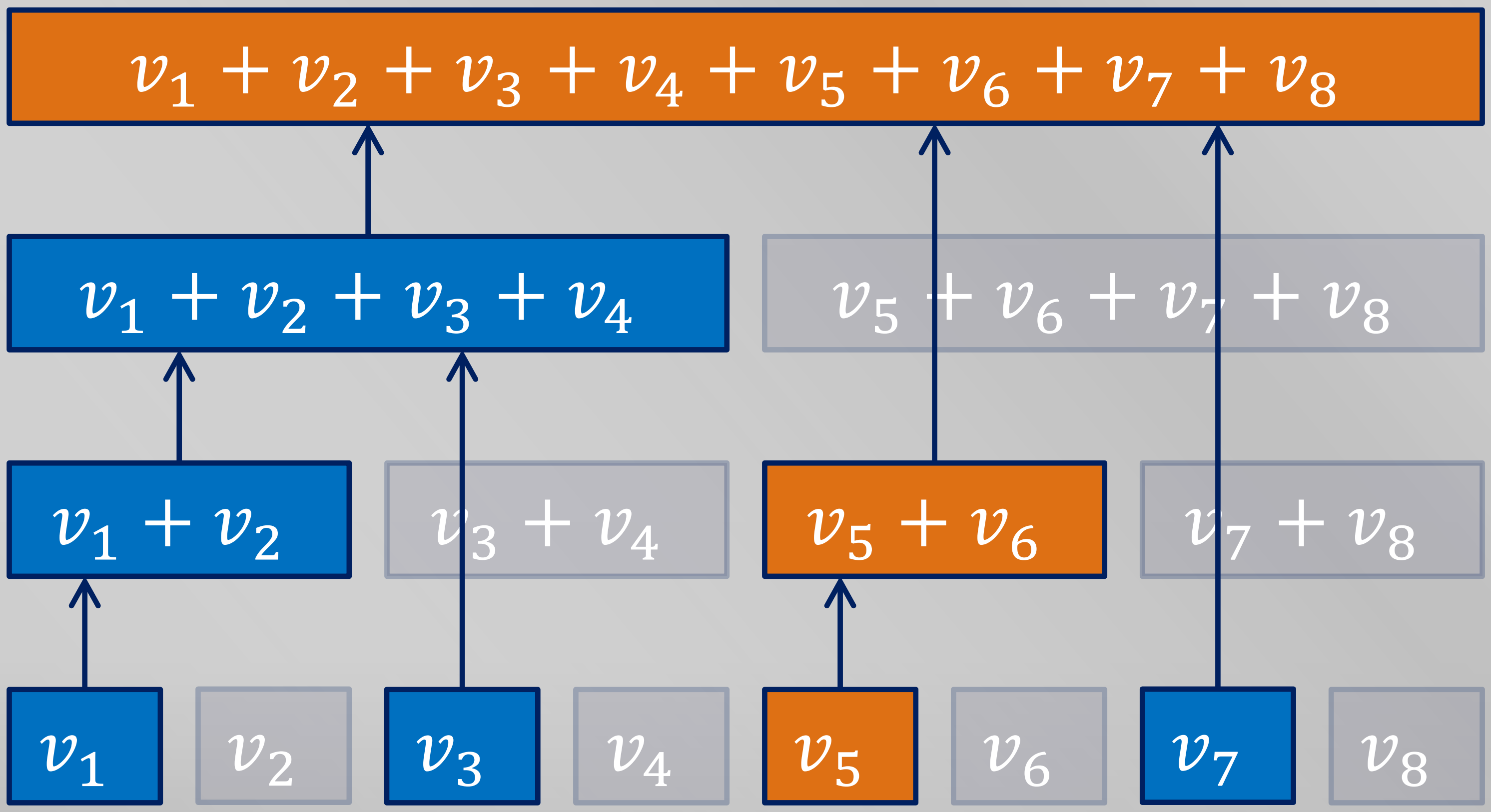


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

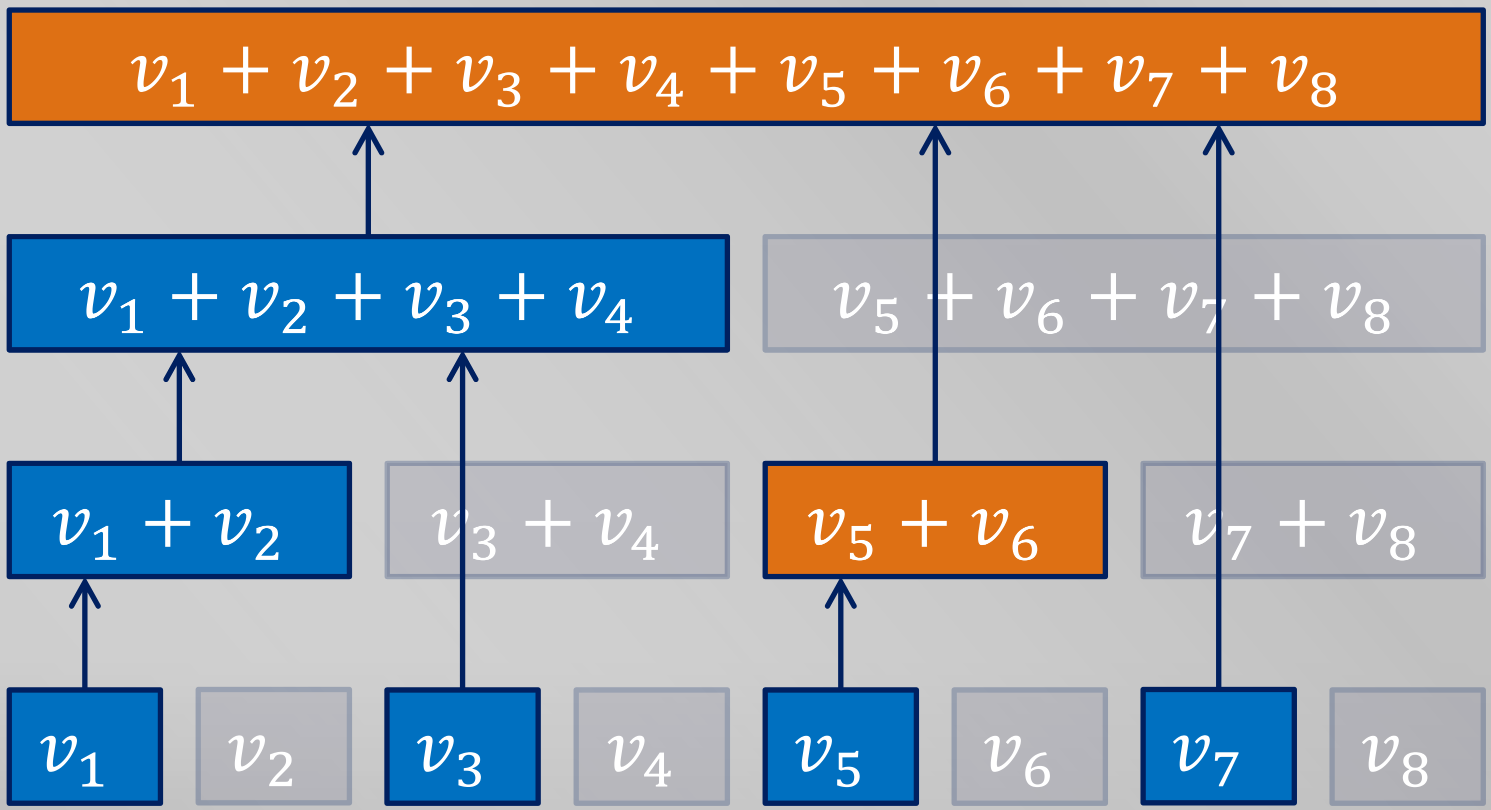
変数の値の更新 (v_4)


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

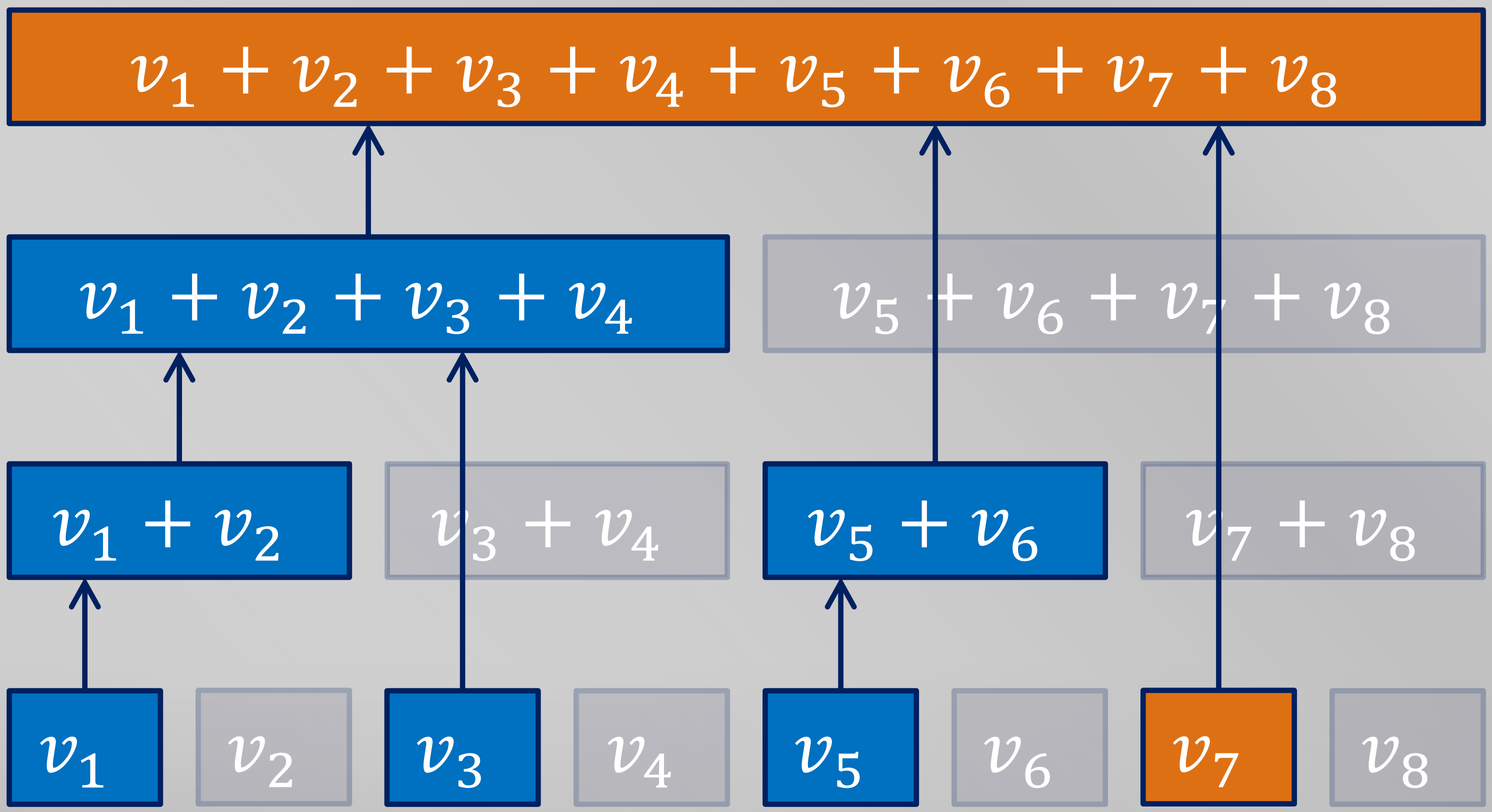
変数の値の更新 (v_5)


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

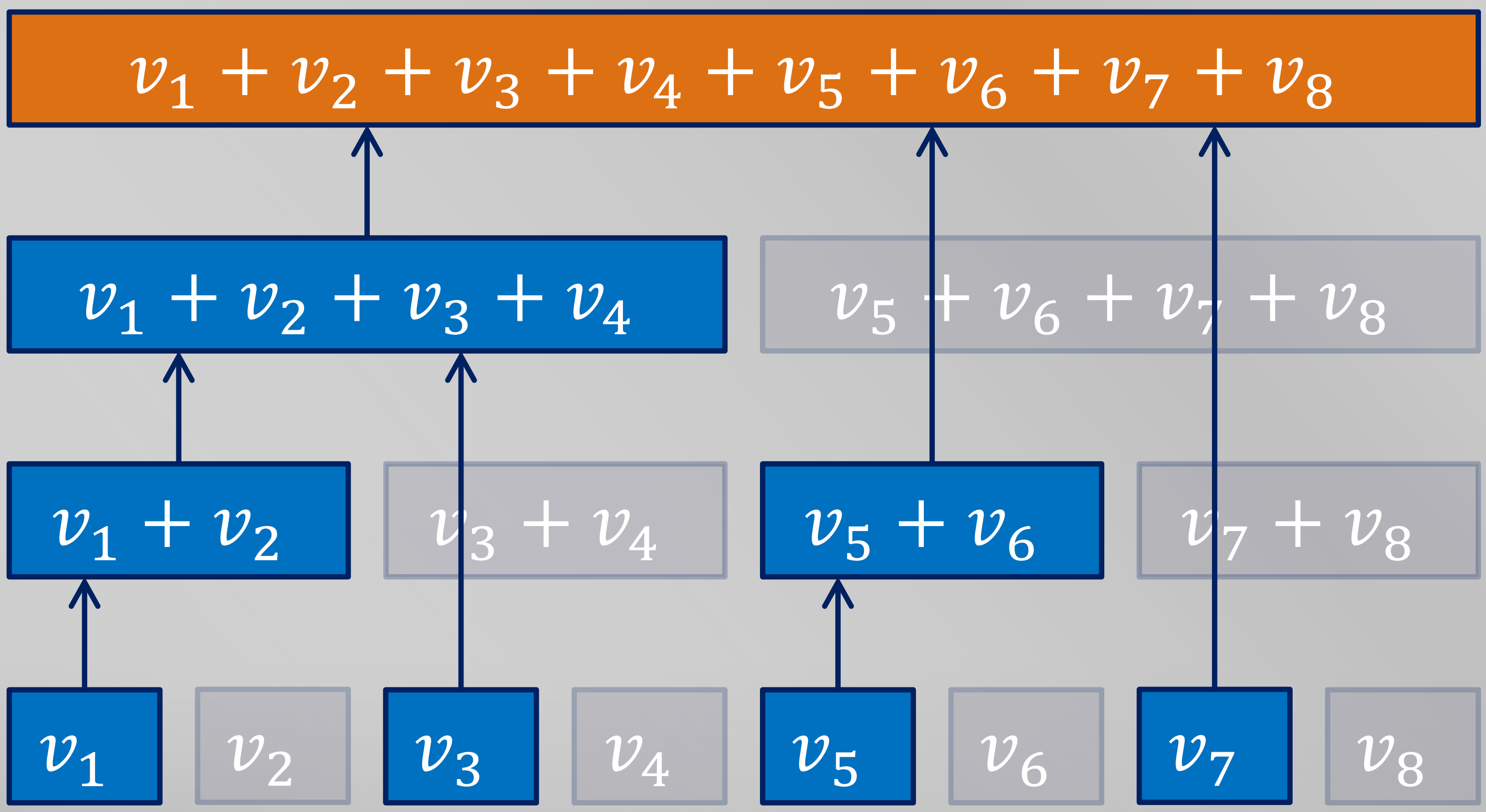
変数の値の更新 (v_6)


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

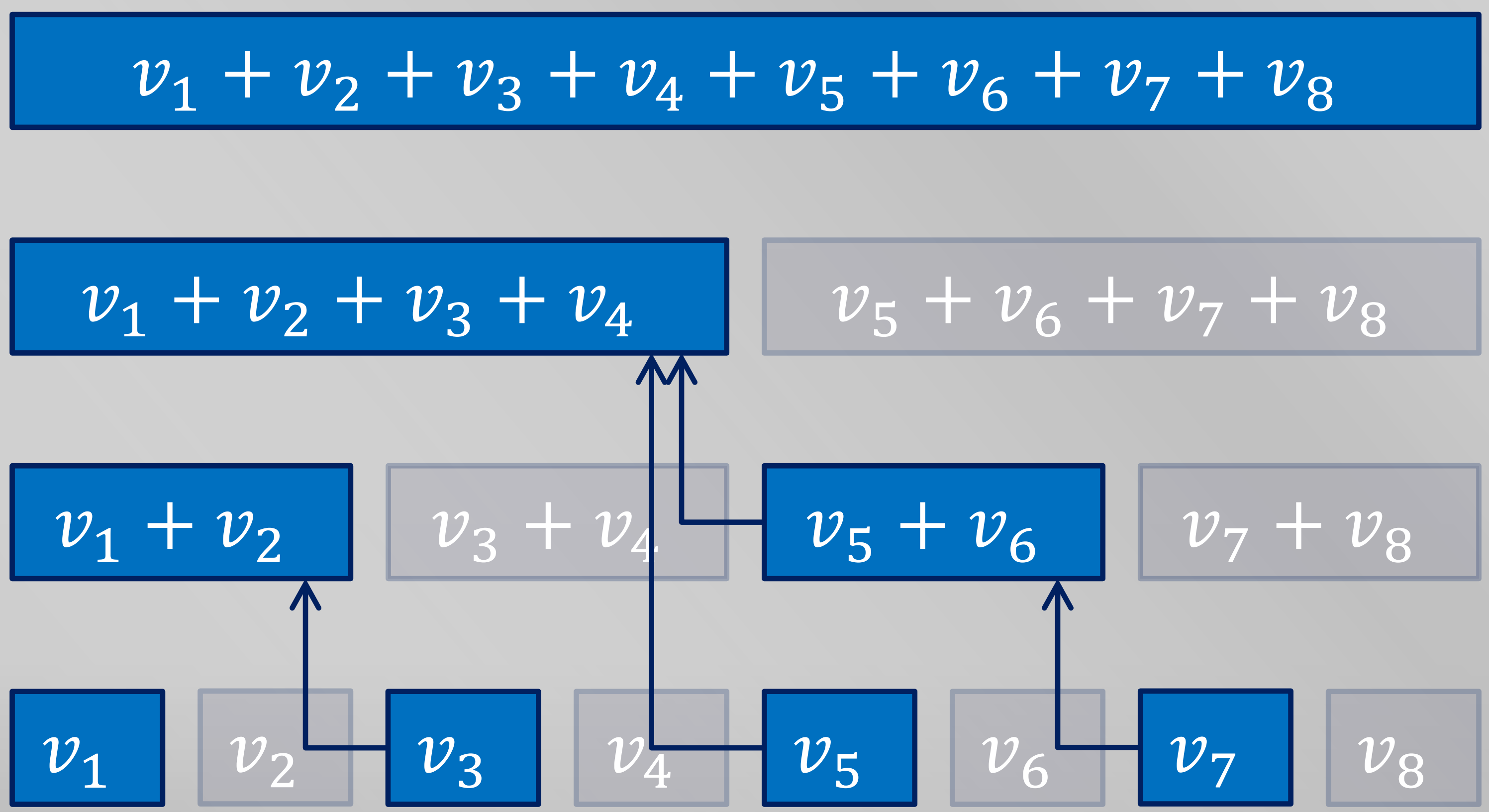
変数の値の更新 (v_7)


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

変数の値の更新 (v_8)

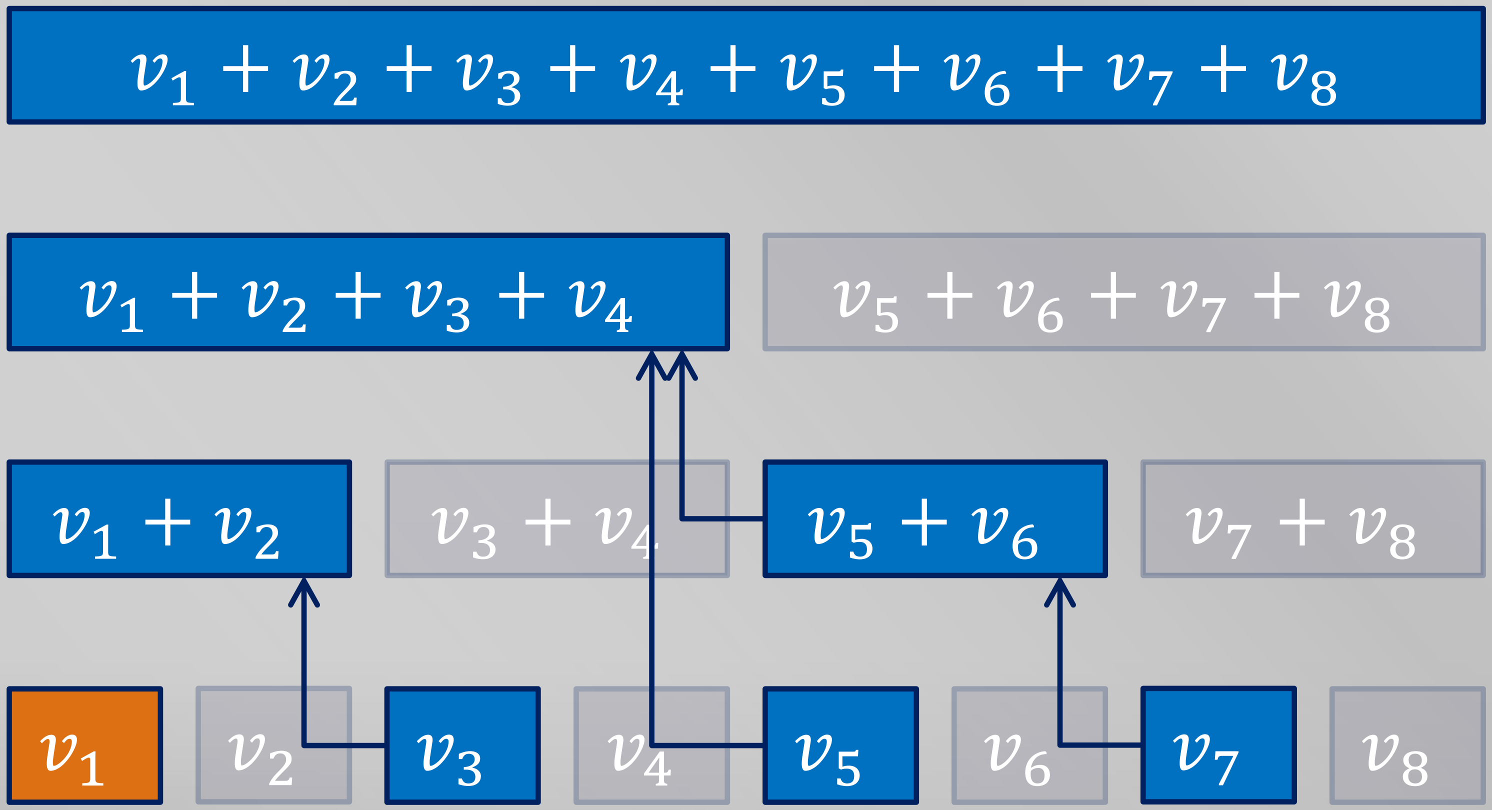

$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

区間の和の計算

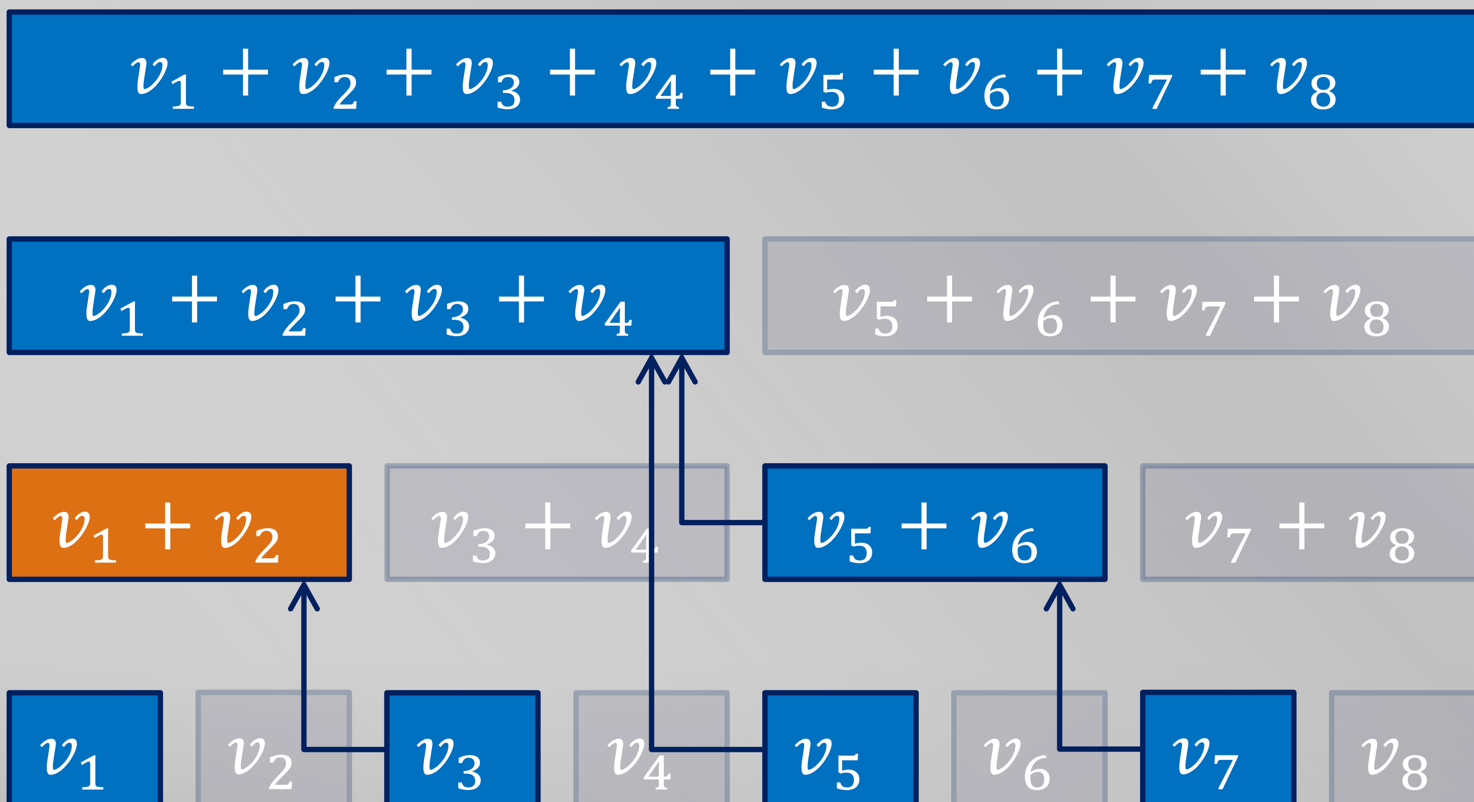


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

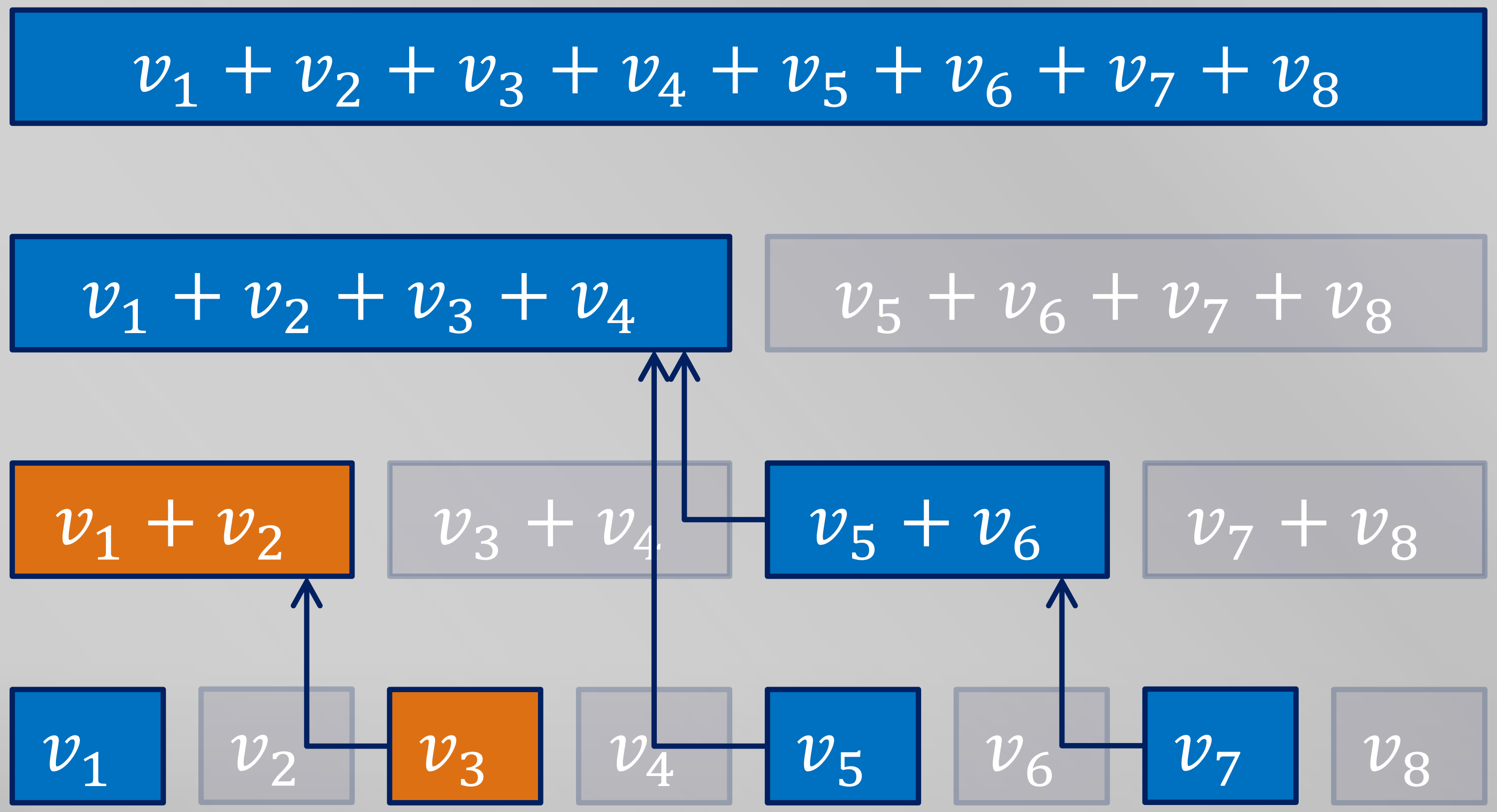
区間の和の計算 $(v_1 + \cdots + v_1)$



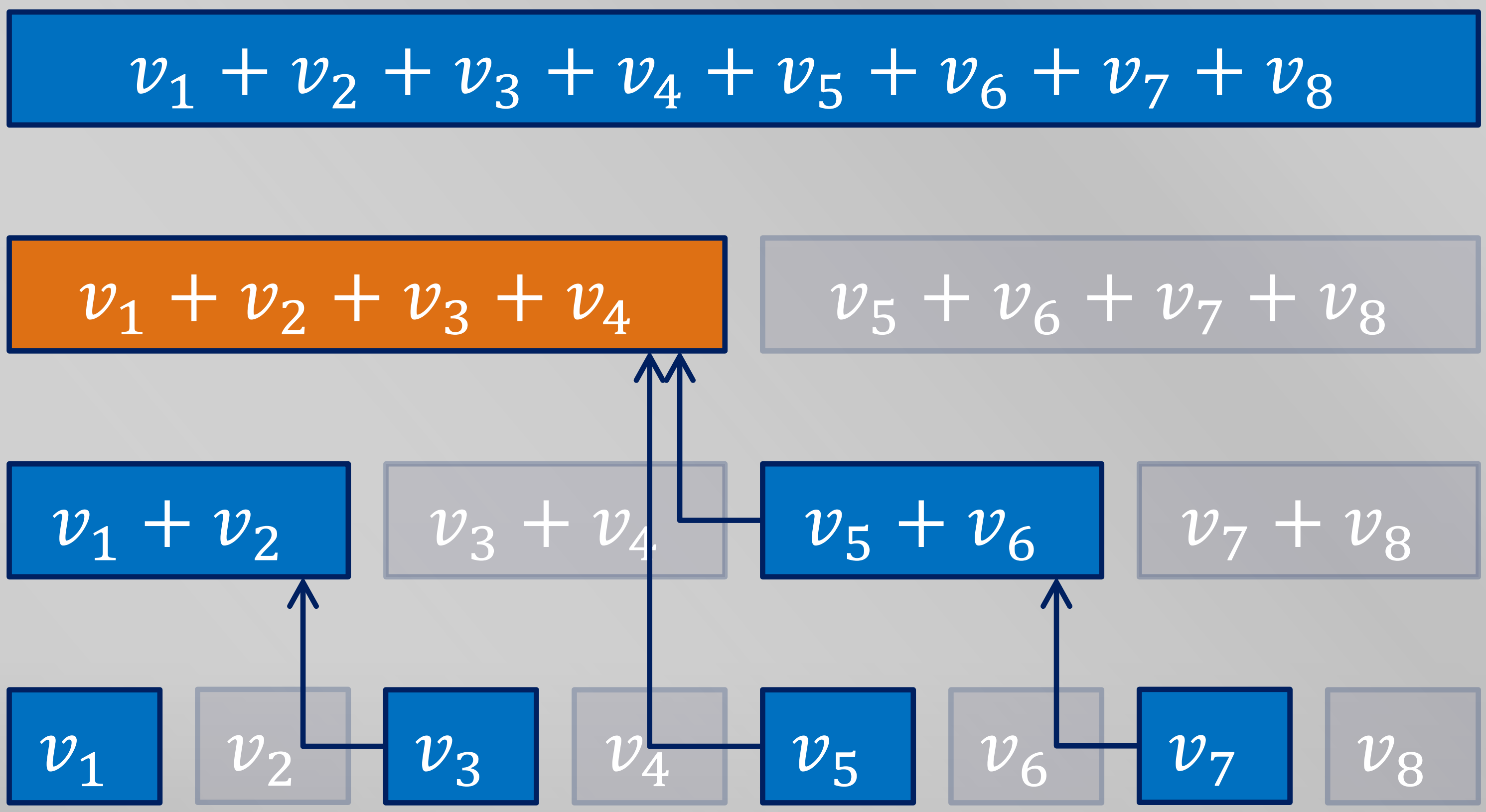
区間の和の計算 ($v_1 + \cdots + v_2$)



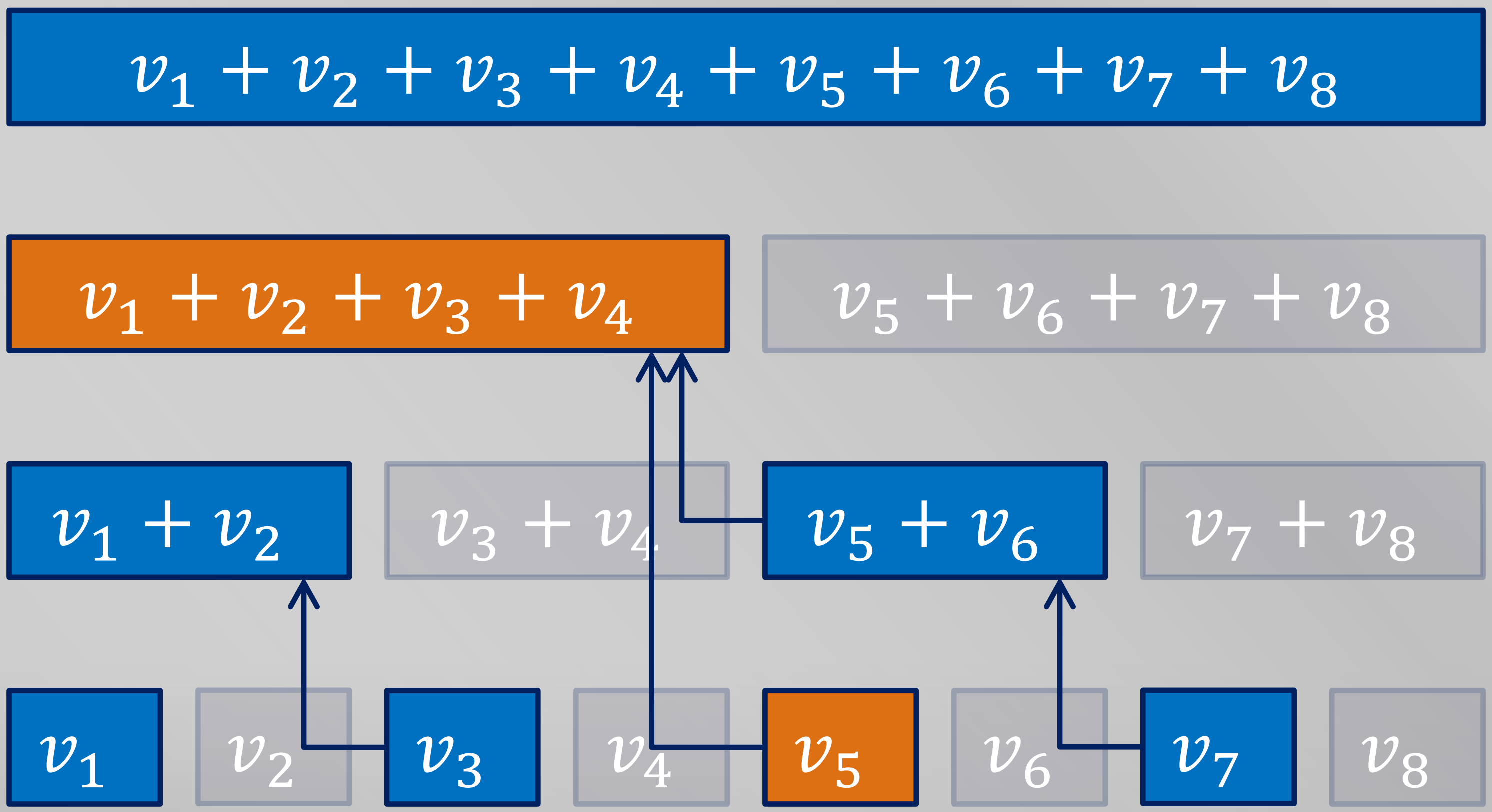
区間の和の計算 $(v_1 + \cdots + v_3)$



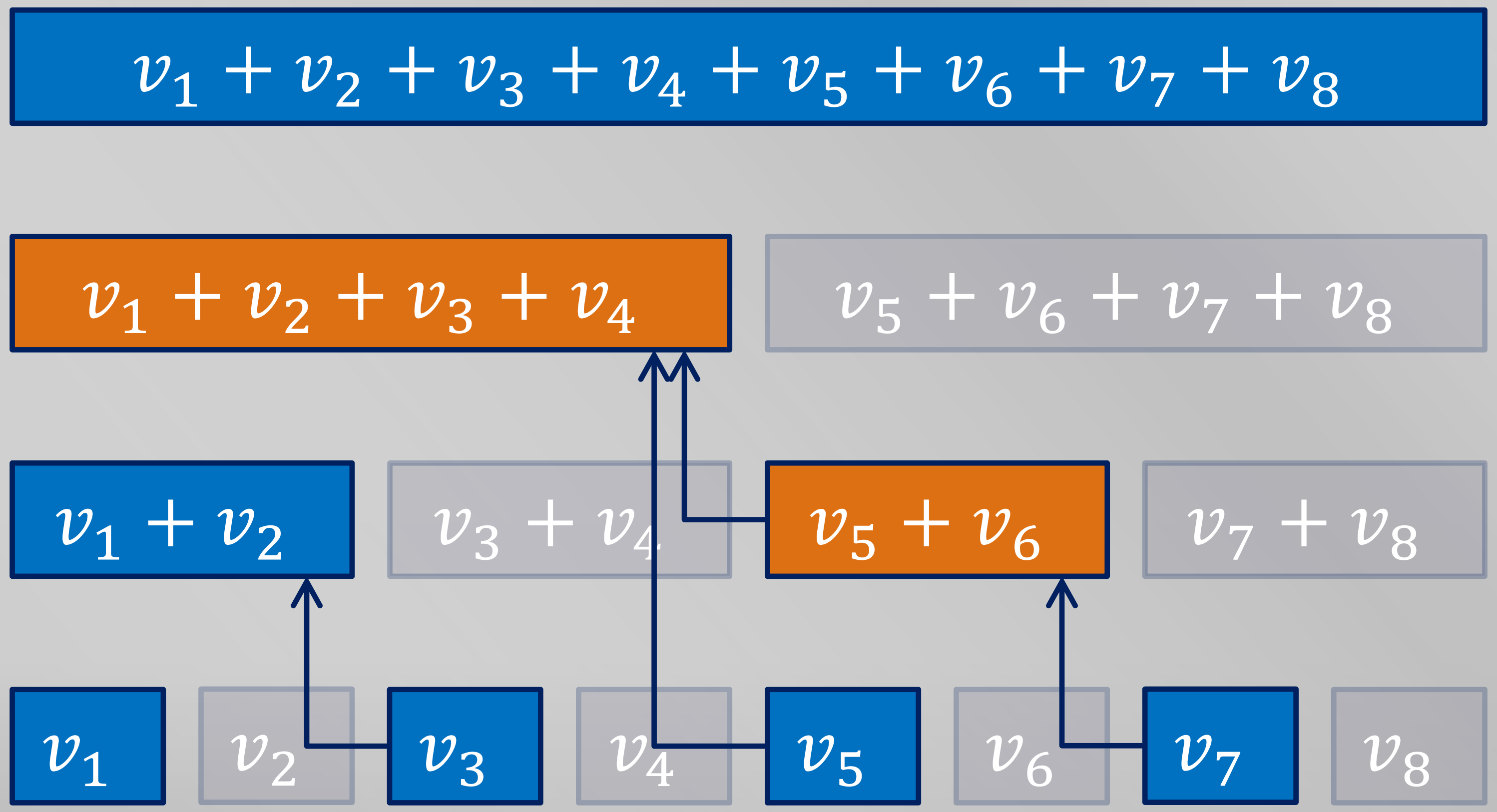
区間の和の計算 $(v_1 + \cdots + v_4)$



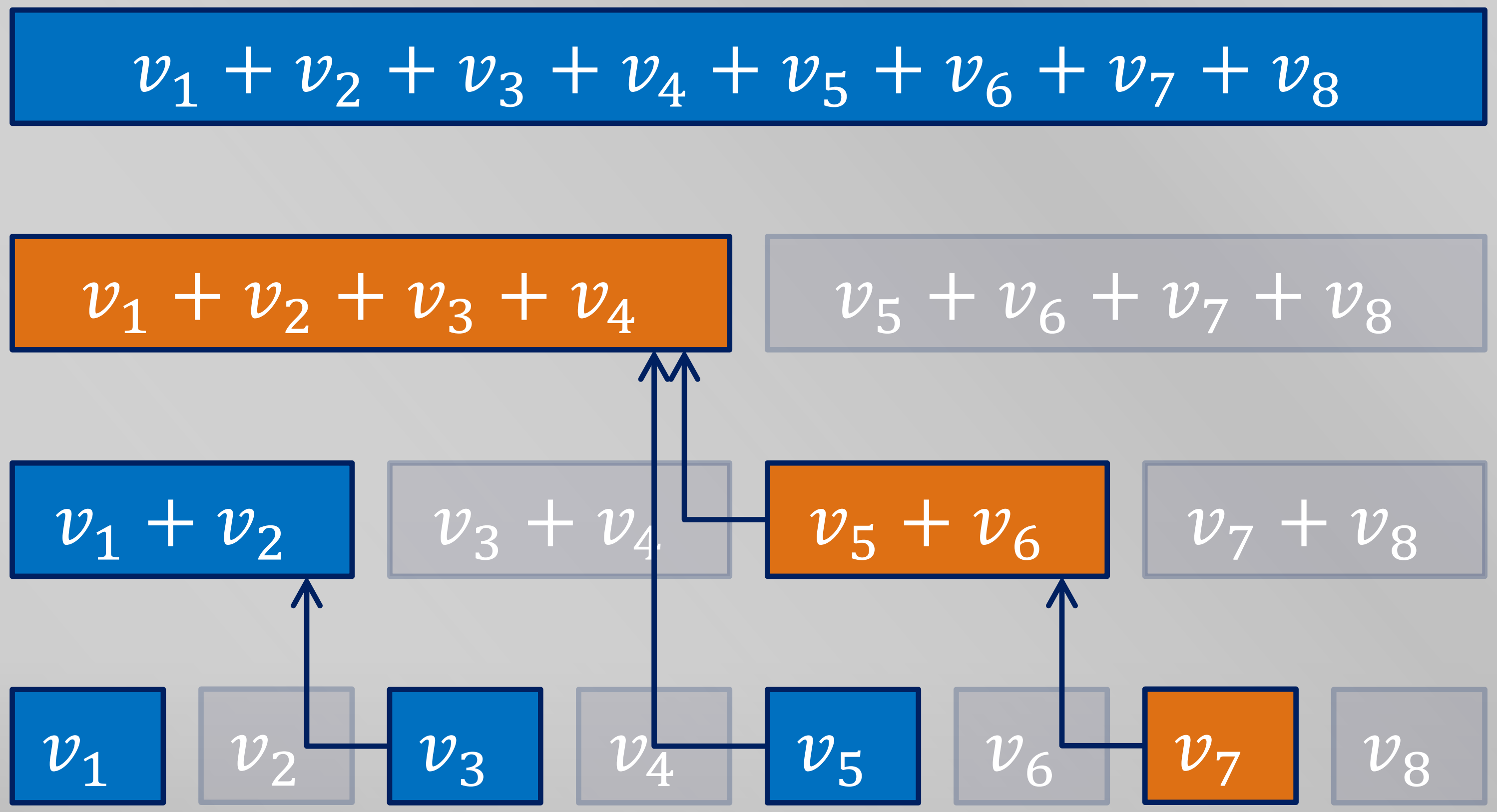
区間の和の計算 ($v_1 + \cdots + v_5$)



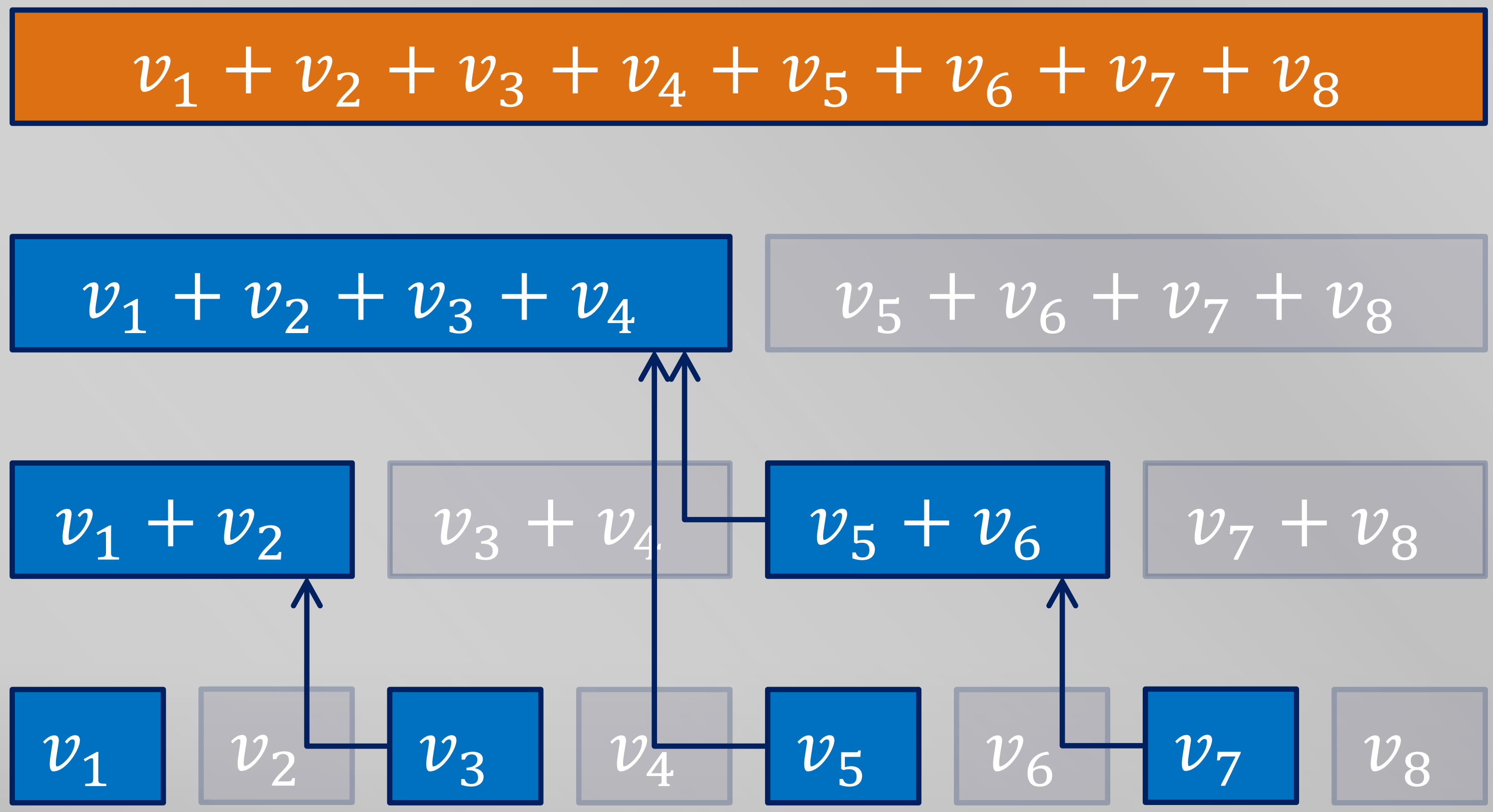
区間の和の計算 $(v_1 + \cdots + v_6)$



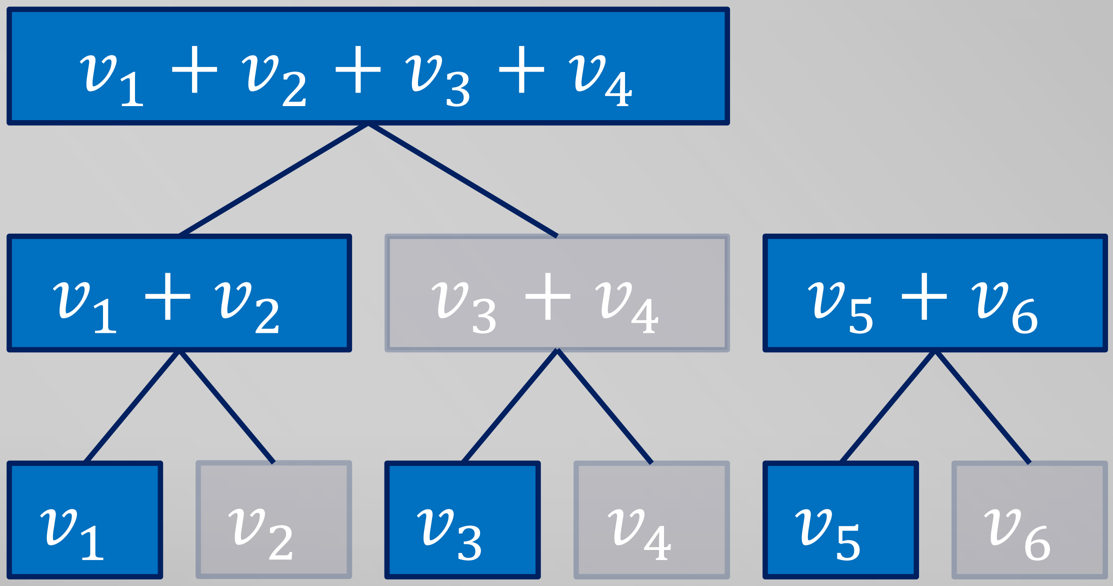
区間の和の計算 $(v_1 + \cdots + v_7)$



区間の和の計算 ($v_1 + \cdots + v_8$)



N は 2 ベキでなくても OK



計算量

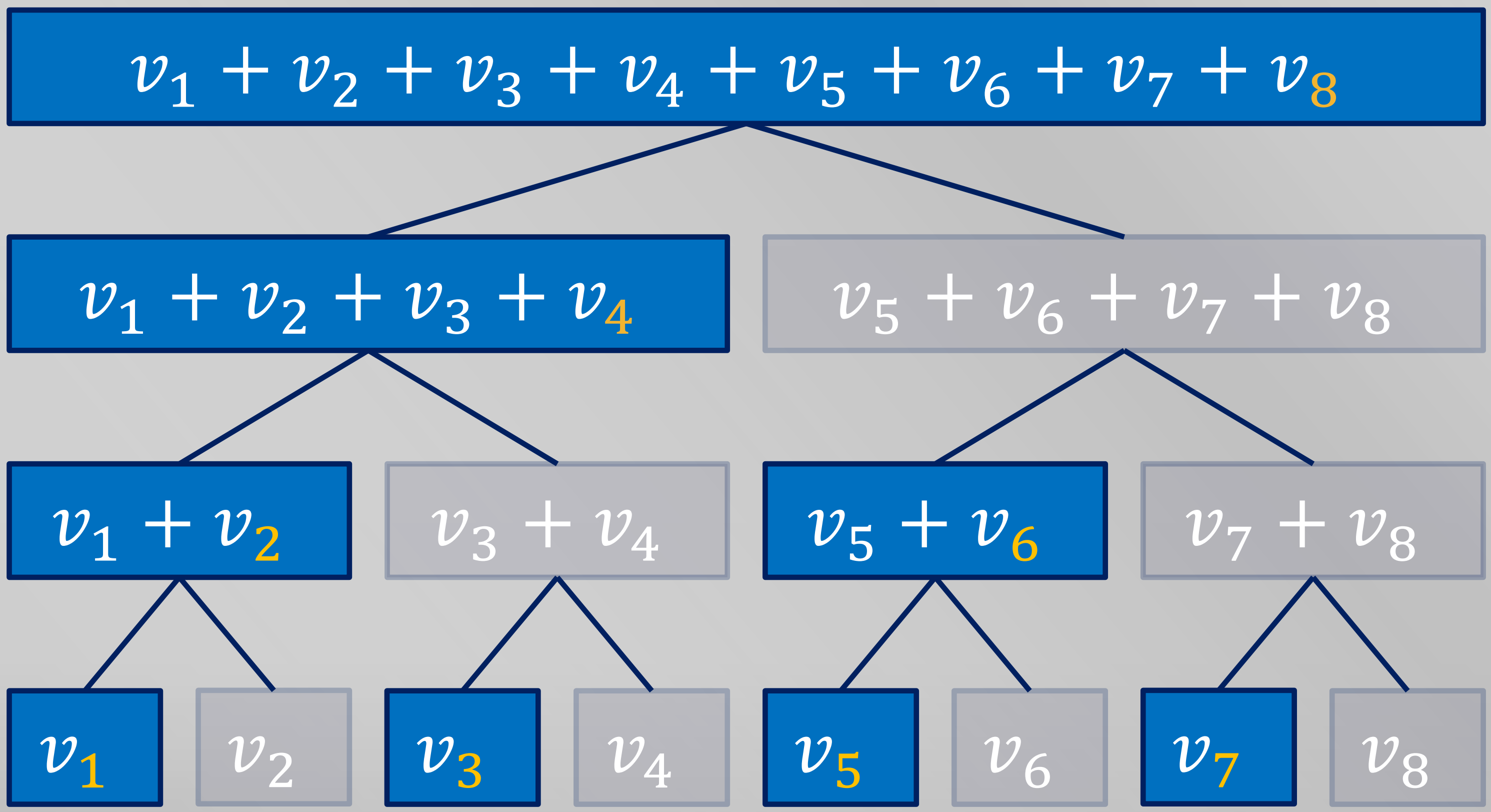
- N 個の区間の和を管理する
  - $O(N)$ メモリ
- 変数の値の更新
  - $O(\log N)$ 時間
    - 高々 $(\log_2 N + 1)$ 個の区間に足す
- prefix の和の計算
  - $O(\log N)$ 時間
    - 高々 $(\log_2 N + 1)$ 個の区間の和

具体的な実装

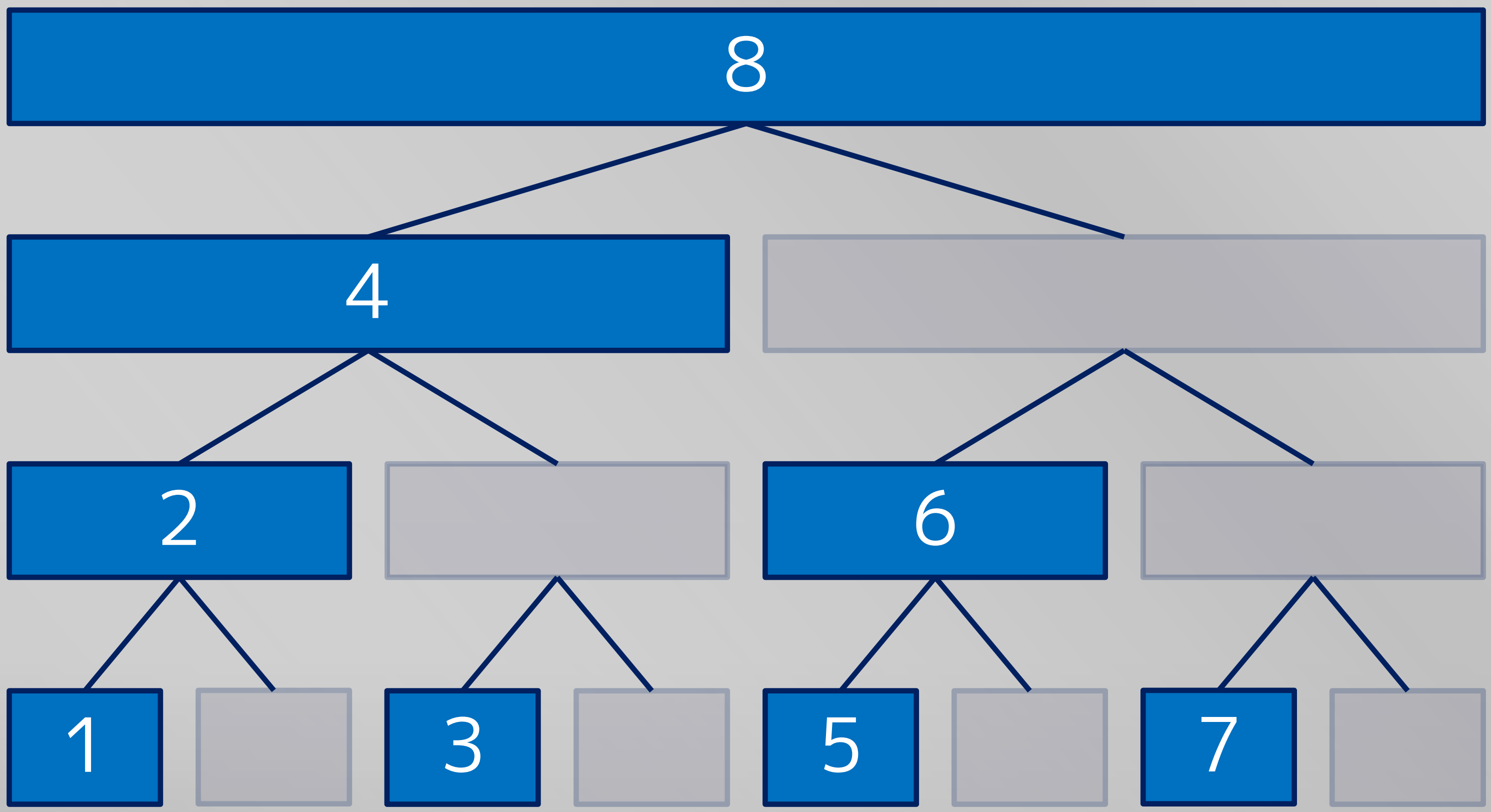
実装例 (C++)

```cpp
int N;
int bit[1000010];
void add(int a, int w) {
  for (int x = a; x <= N; x += x & -x) bit[x] += w;
}
int sum(int a) {
  int ret = 0;
  for (int x = a; x > 0; x -= x & -x) ret += bit[x];
  return ret;
}
```

区間の右端で番号づけ


$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8$
$v_1 + v_2 + v_3 + v_4$
$v_5 + v_6 + v_7 + v_8$
$v_1 + v_2$
$v_3 + v_4$
$v_5 + v_6$
$v_7 + v_8$
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8

区間の右端で番号づけ

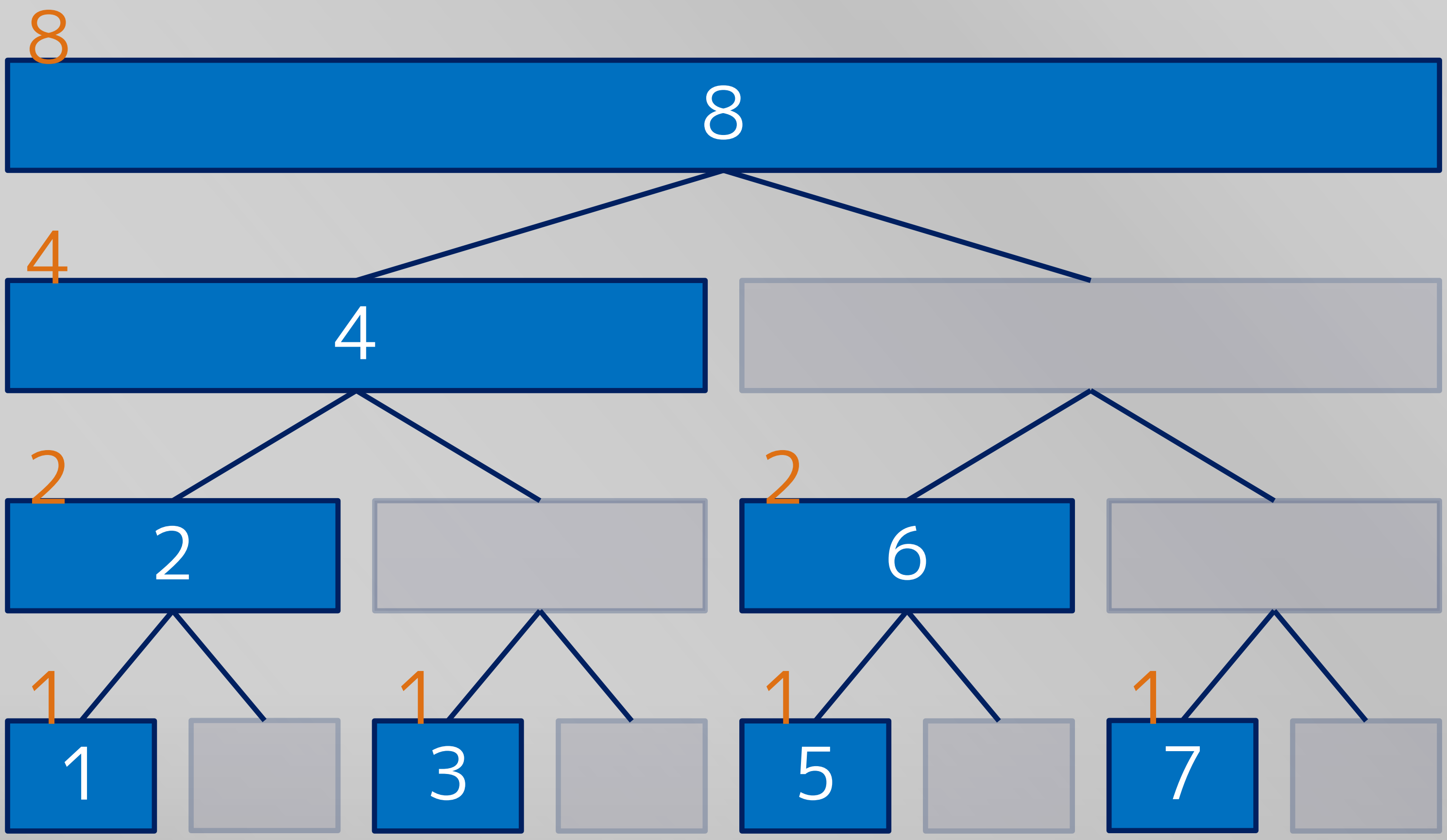


区間の右端で番号づけ

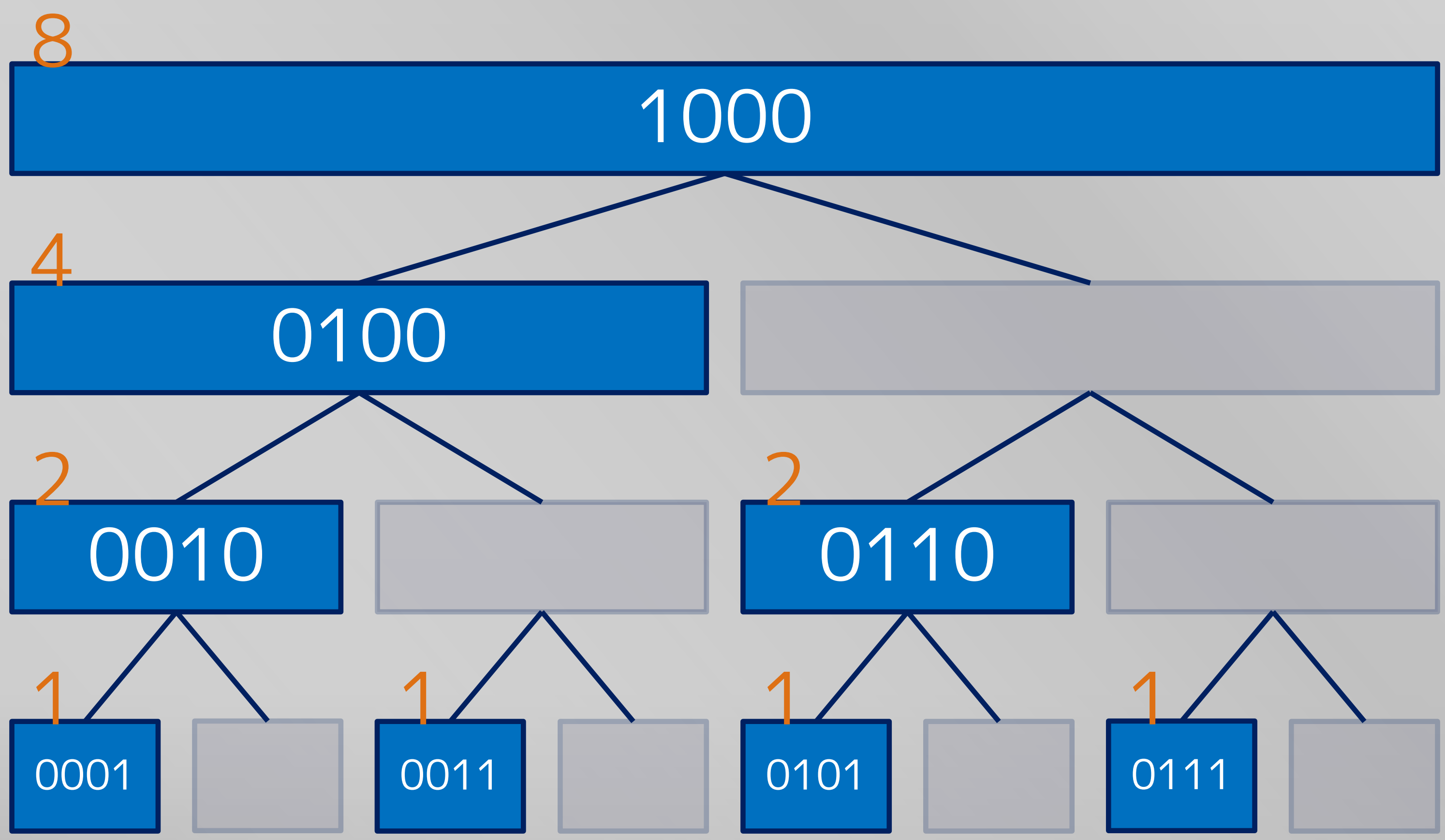
- bit[1] から bit[N] までを使用

```
int N;
int bit[1000010];
```

区間の長さと番号



区間の長さと番号を二進数で見る



区間の長さと番号

- bit[x] が管理する区間の長さは，x の最も下の立っているビット

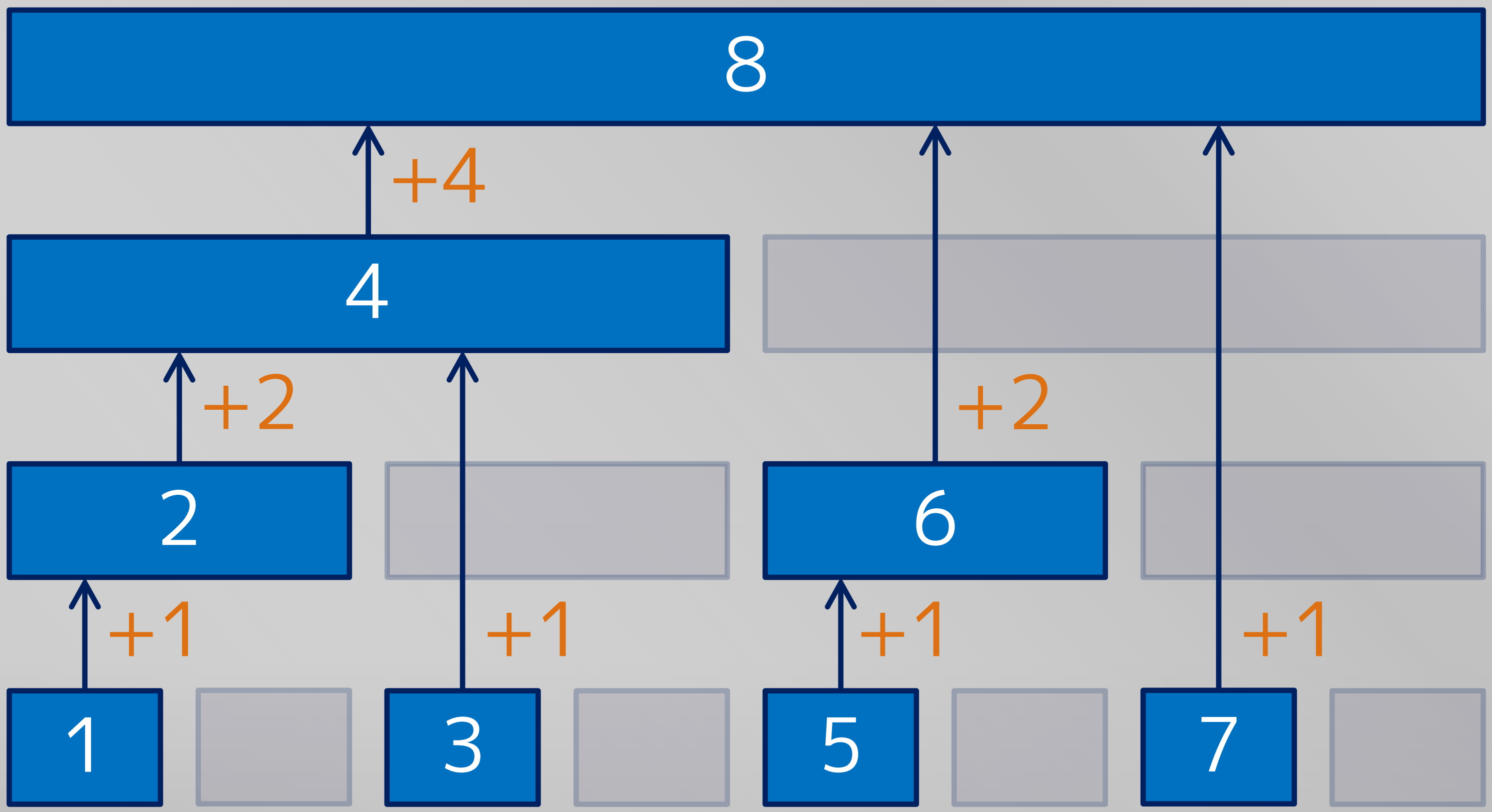
x & -x

区間の長さと番号

- x の最も下の立っているビットは，x & -x で取り出せる
  - 覚えてしまいましょう

```
     x = 00000000 00000000 00101110 01011000
    -x = 11111111 11111111 11010001 10101000
x & -x = 00000000 00000000 00000000 00001000
```

変数の値の更新

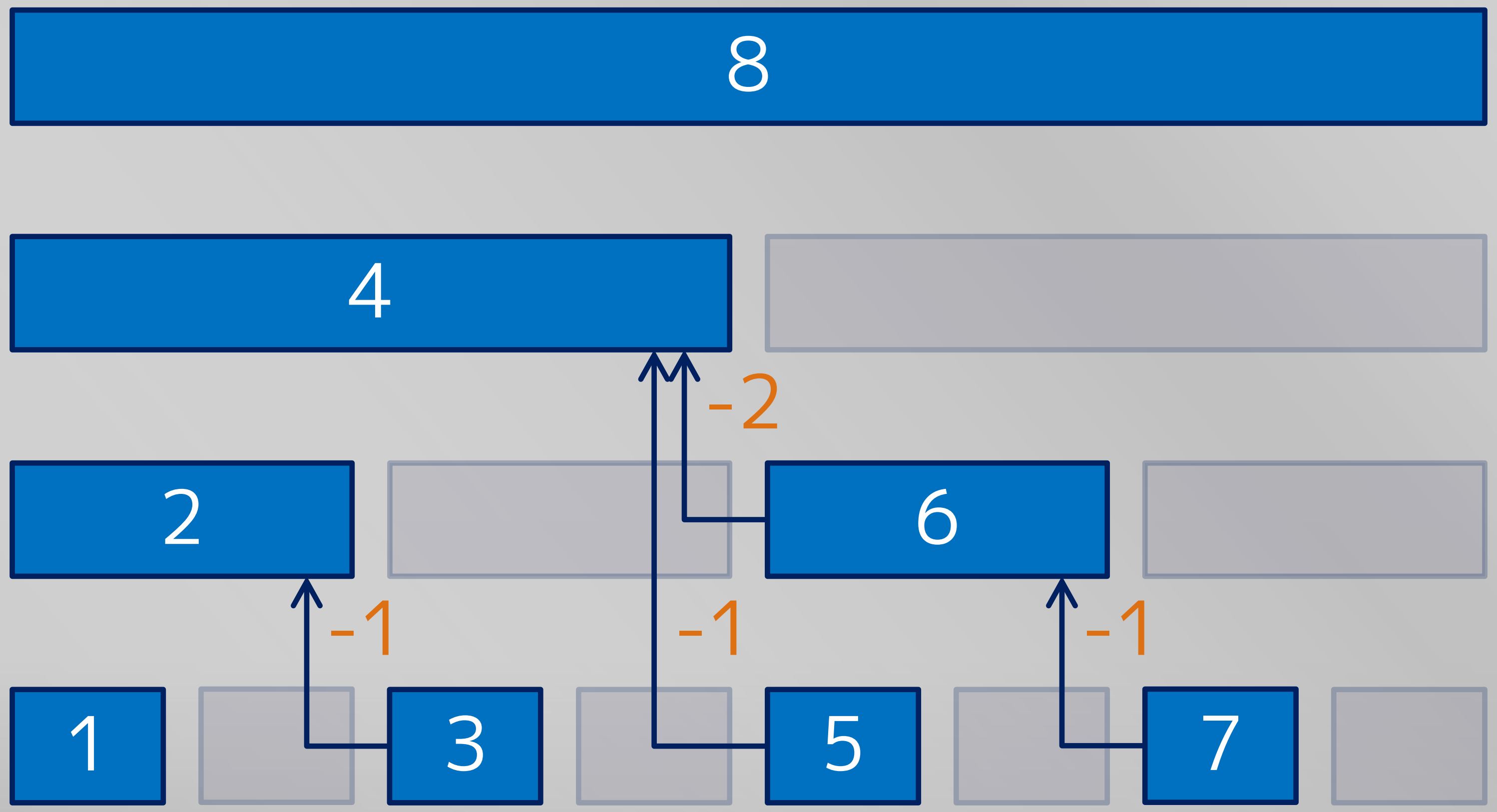


変数の値の更新

- 次に更新すべき区間は，番号に区間の長さを足すと求まる

```
// v[a] += w
void add(int a, int w) {
  for (int x = a; x <= N; x += x & -x) bit[x] += w;
}
```

区間の和の計算



区間の和の計算

- 次に足すべき区間は，番号から区間の長さを引くと求まる

```
// v[1] + ... + v[a]
int sum(int a) {
  int ret = 0;
  for (int x = a; x > 0; x -= x & -x) ret += bit[x];
  return ret;
}
```

完成！

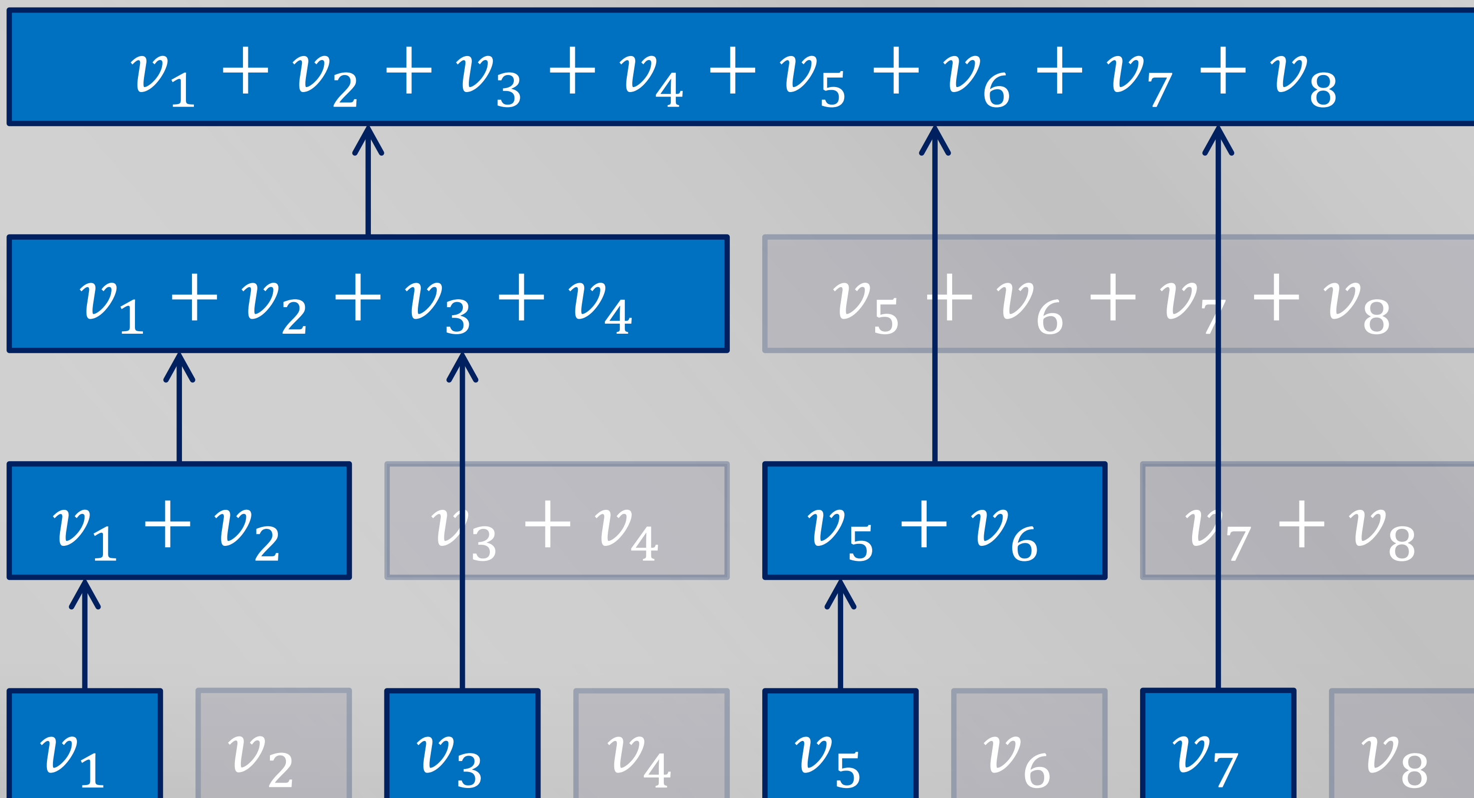
```
int N;
int bit[1000010];
void add(int a, int w) {
  for (int x = a; x <= N; x += x & -x) bit[x] += w;
}
int sum(int a) {
  int ret = 0;
  for (int x = a; x > 0; x -= x & -x) ret += bit[x];
  return ret;
}
```

0 以外の値で初期化

- add を N 回呼び出せば $O(N \log N)$ 時間
  - ほとんどの場合これで十分だと思います
- $v_x = 1$ で初期化するなら `bit[x] = x & -x`
- 一般には bit[x] を v_x で初期化したのち

```
for (int x = 1; x < N; ++x) bit[x + (x & -x)] += bit[x];
```

この木で累積和をとっている感じ



添え字を 0 から始めたいあなたへ

- 添え字を「1 から N まで」の代わりに「0 から $N-1$ まで」にしたいこともある
  - 毎回 1 を足したり引いたりをかませてもいいけれど結構な混乱の元です
  - というわけで番号から 1 を引いたときのリンクを辿る式を紹介
    - 式変形の見通しは悪くなりますが，動きを把握していれば丸暗記でもよいでしょう
  - 本講義ではここ以外は BIT の添え字は 1 からです

添え字を 0 から始めたいあなたへ

```
// v[a] += w
void add(int a, int w) {
  for (int x = a; x < N; x |= x + 1) {
    bit[x] += w;
  }
}
```

添え字を 0 から始めたいあなたへ

```
// v[0] + ... + v[a - 1]
int sum(int a) {
  int ret = 0;
  for (int x = a - 1; x >= 0; x = (x & (x + 1)) - 1) {
    ret += bit[x];
  }
  return ret;
}
```

応用範囲

基本的な問題

- N 個の変数 $v_1, \dots, v_N$
  - すべて 0 で初期化
- 2 種類のクエリ
  - v_a に値 w を加える
  - prefix $[1, a]$ のところの和 $v_1 + v_2 + \cdots + v_a$ を求める
- クエリあたり $O(\log N)$ 時間にしたい

和でなくても OK

- N 個の変数 $v_1, \ldots, v_N$
  - すべて 0 で初期化
- 2 種類のクエリ
  - v_a を値 w に変更 (ただし $v_a \leqq w$)
  - prefix $[1, a]$ のところの最大値 $\max\{v_1, v_2, \ldots, v_a\}$ を求める
- クエリあたり $O(\log N)$ 時間にしたい

和でないときにできないこと

- 小さい値に更新することはできない
- prefix 以外の区間の max は一般にはわからない
  - $v_a + v_{a+1} + \cdots + v_b = (v_1 + \cdots + v_b) - (v_1 + \cdots + v_{a-1})$ だから，和に関しては prefix の和さえわかれば他の区間についてもわかる
- 和以外は無理せず Segment Tree を用いるのも十分あり
  - BIT はどうしても速度・メモリがきついとき用に

区間に対する更新

- BIT でできることは「1 点の更新」「区間の和など」
  - 「区間に対する更新」などには基本的に何らかの式変形が必要になると考えてよいでしょう

区間に対する更新

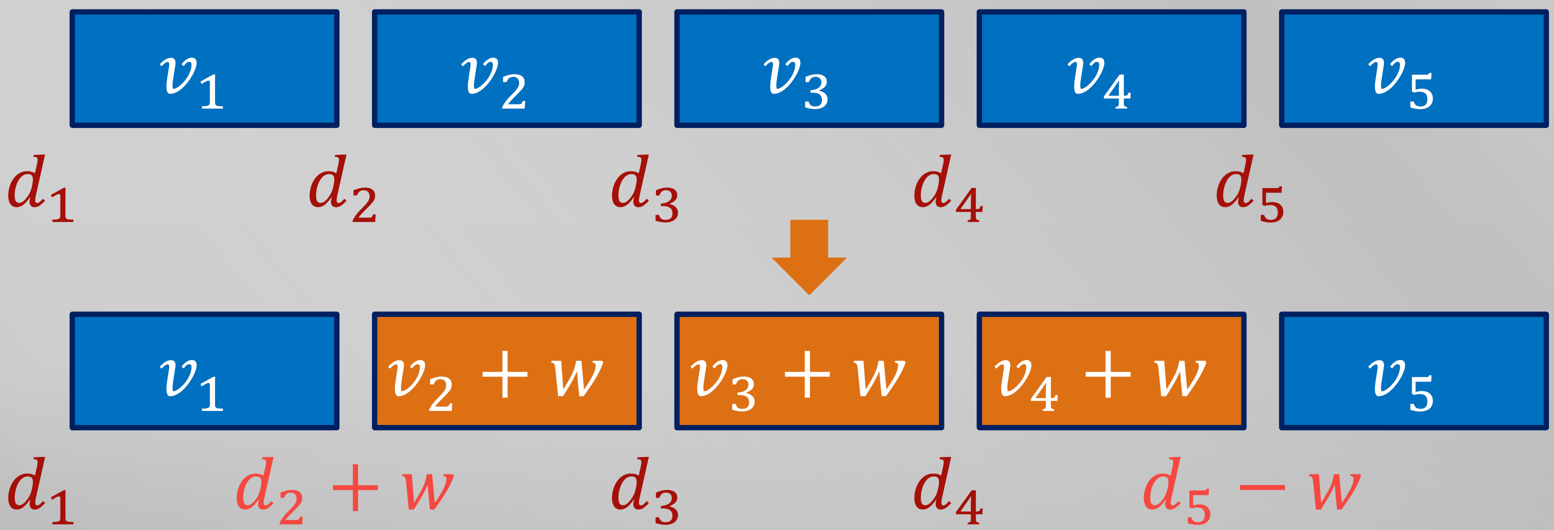
- N 個の変数 $v_1, \dots, v_N$
  - すべて 0 で初期化
- 2 種類のクエリ
  - 区間 $[a, b]$ のところ $v_a, v_{a+1}, \dots, v_b$ に値 w を加える
  - v_a の値を求める
- クエリあたり $O(\log N)$ 時間にしたい

差分に注目

- $d_x = v_x - v_{x-1}$ とおく
  - ただし $v_0 = 0$ と考える

差分に注目

- 区間 $[a, b]$ のところ $v_a, v_{a+1}, \ldots, v_b$ に値 w を加える
  - d_a に w を，d_{b+1} に $-w$ を加える



差分に注目

- v_a の値を求める
  - 和 $d_1 + d_2 + \cdots + d_a$ を求める

差分に注目

- N 個の変数 $d_1, \ldots, d_N$
  - すべて 0 で初期化
- 2 種類のクエリ
  - d_a に w を，d_{b+1} に $-w$ を加える
  - 和 $d_1 + d_2 + \cdots + d_a$ を求める
- BIT でできる！

区間に対する更新＆区間の和

- N 個の変数 $v_0, \dots, v_{N-1}$
  - すべて 0 で初期化
- 2 種類のクエリ
  - 区間 $[a, b)$ のところ $v_a, v_{a+1}, \dots, v_{b-1}$ に値 w を加える
  - 区間 $[0, c)$ のところの和 $v_0 + v_1 + \cdots + v_{c-1}$ を求める
- クエリあたり $O(\log N)$ 時間にしたい

区間に対する更新＆区間の和

1. Segment Tree に逃げる
   - いいと思います
2. かしこい式変形を用いる
3. かしこくない式変形を用いる

かしこい式変形

- 変数 $p_0, p_1, \dots, p_N, q_0, q_1, \dots, q_N$ をとる
- 2 種類のクエリ
  - 区間 $[a, b)$ のところ $v_a, v_{a+1}, \dots, v_{b-1}$ に値 w を加える
    - p_a に $-wa$ を，p_b に wb を，q_a に w を，q_b に $-w$ を加える
  - 区間 $[0, c)$ のところの和 $v_0 + v_1 + \cdots + v_{c-1}$ を求める
    - $(p_0 + p_1 + \cdots + p_c) + (q_0 + q_1 + \cdots + q_c)c$ を求める
- BIT を 2 個使えばできる

かしこい式変形の説明 (略)

- 和なので 1 クエリ分正しければ OK
- 2 種類のクエリ
  - 区間 $[a, b)$ のところ $v_a, v_{a+1}, \dots, v_{b-1}$ に値 w を加える
    - p_a に $-wa$ を，p_b に wb を，q_a に w を，q_b に $-w$ を加える
  - 区間 $[0, a)$ のところの和 $v_0 + v_1 + \cdots + v_{a-1}$ を求める
    - $(p_0 + p_1 + \cdots + p_c) + (q_0 + q_1 + \cdots + q_c)c$ を求める
- $c < a$, $a \leqq c < b$, $b \leqq c$ のそれぞれの場合を計算してみよう

かしこくない式変形

- 部分和をとる
- 0 次の係数と 1 次の係数に対応する BIT は作る
- 結局先ほどの式変形になります

2 次元の問題

- $M \times N$ 個の変数 $v_{x,y}$ $(x = 1, \dots, M,\ y = 1, \dots, N)$
  - すべて 0 で初期化
- 2 種類のクエリ
  - $v_{a,b}$ に値 w を加える
  - $[1, a] \times [1, b]$ のところの和 $\sum_{1 \leqq x \leqq a, 1 \leqq y \leqq b} v_{x,y}$ を求める
- クエリあたり $O((\log M)(\log N))$ 時間にしたい

2 次元の問題

- BIT が BIT をもつ感じ
- `bit[x][y]` に和 $\sum_{L_x < i \leqq x, L_y < j \leqq y} v_{i,j}$ をもたせる
  - ただし L_x, L_y は `x - (x & -x)`, `y - (y & -y)`
- 実装は単純な 2 重ループ
  - Segment Tree 等と比べてはっきり優れていると思います
- 3 次元以上も同じ

実装例 (C++)

```cpp
int N;
int bit[1010][1010];
void add(int a, int b, int w) {
  for (int x = a; x <= M; x += x & -x)
    for (int y = b; y <= N; y += y & -y) {
      bit[x][y] += w;
    }
  }
}
```

実装例 (C++)

```cpp
int sum(int a, int b) {
  int ret = 0;
  for (int x = a; x > 0; x -= x & -x) {
    for (int y = b; y > 0; y -= y & -y) {
      ret += bit[x][y];
    }
  }
  return ret;
}
```

BIT 上で二分探索

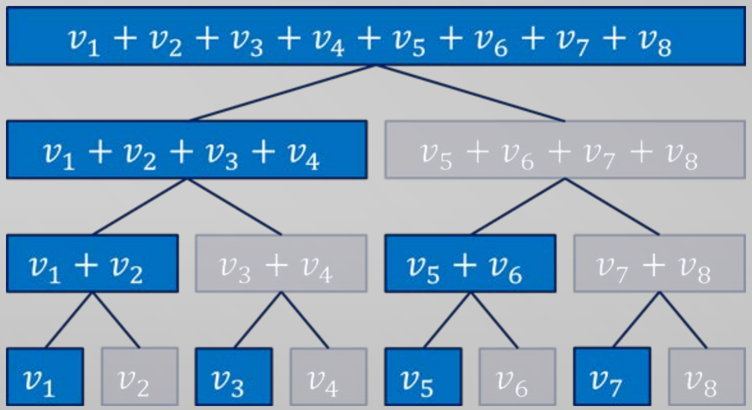
- N 個の変数 $v_1, \ldots, v_N$
- 3 種類のクエリ
  - v_a に値 w を加える (ただし常に $v_a \geqq 0$ が成り立つとする)
  - prefix $[1, a]$ のところの和 $v_1 + v_2 + \cdots + v_a$ を求める
  - $v_1 + v_2 + \cdots + v_x \geqq w$ となる最小の x を求める
- クエリあたり $O(\log N)$ 時間にしたい

普通に二分探索

- $v_1 + v_2 + \cdots + v_x$ は
  - x について単調なので，二分探索で w 以上となる最小の場所がわかる
  - それぞれ $O(\log N)$ 時間で計算できる
- $O((\log N)^2)$ 時間

BIT 上で二分探索

- 二分木の分かれ方に従って二分探索する
- 左の子に進むか右の子に進むかを知るためには，左の子の範囲の和がわかればよい
  - ちょうど BIT がもっている情報，$O(1)$ 時間で得られる



BIT 上で二分探索

```
int lowerBound(int w) {
  if (w <= 0) return 0;
  int x = 0;
  for (int k = (n 以下の最小の 2 べキ); k > 0; k /= 2) {
    if (x + k <= N && bit[x + k] < w) {
      w -= bit[x + k];
      x += k;
    }
  }
  return x + 1;
}
```

BIT 上で二分探索できると嬉しいこと

- 変数の値を $0, 1$ として考えると，集合への要素の追加・削除，「指定された要素が何番目に小さいか」「w 番目に小さい要素は何か」ができる
  - 値の範囲がわかっている (1 から N まで) 場合の `std::set` より高機能なもの
    - メモリは $O(N)$ かかる
    - 座標圧縮して使うことも多い

まとめ

- 基本
  - 1 点に足す・prefix の和を求める
  - bit[x] に右端が x で長さ x & -x の区間の和をもたせる
- 応用
  - 差分・部分和に対する問題を考えてみる
  - 多次元は多重ループ
  - 高速に二分探索できる

http://hos.ac/slides/