アルゴリズムとデータ構造
第4回: 探索問題(2)

担当: 上原隆平 (uehara)

内容

- 2分探索法
- ハッシュ法

- オーダ記法（バッハマン-ランダウ 記法）
  - ビッグオー記法: O(f(n))
  - ビッグオメガ記法: Ω(f(n))
  - シータ記法: Θ(f(n))

Paul Bachmann
1837−1920

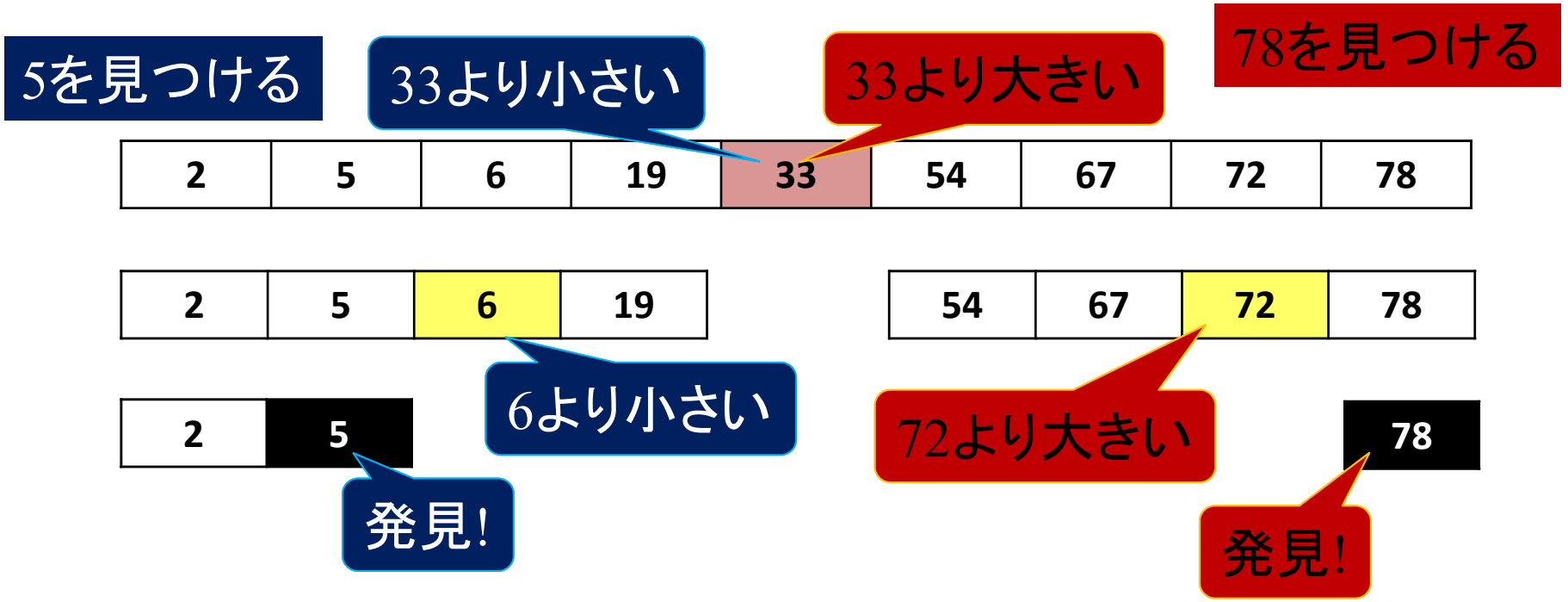
Edmund Landau
1877−1938

Binary search

2分探索法

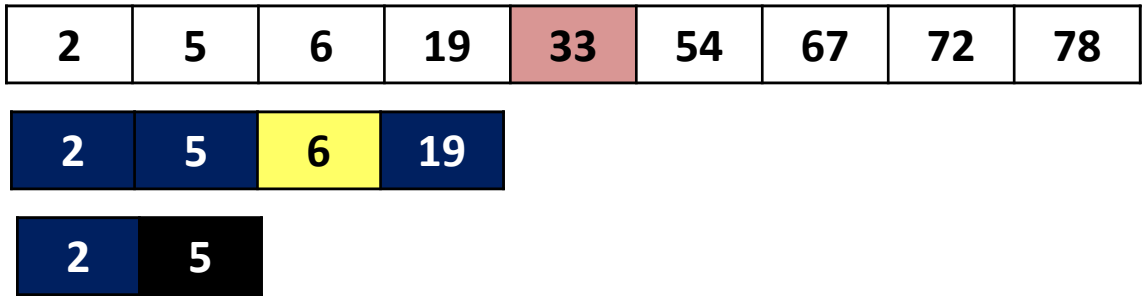
2分探索法

- ソートされた探索空間を二つに分けて探索



  - 分けるポイントはだいたい真ん中にすると良い

2分探索法



- 探索範囲[left, right]の中央の値 s[mid] と探したい値を比較
  - x > 中央の値 ➔ 探索を右半分に限定できる
    left = mid+1; rightは同じ
  - x < 中央の値 ➔ 探索を左半分に限定できる
    leftは同じ，right = mid-1
  - x = 中央の値 ➔ 見つかった
- 以上を探索範囲がなくなるまで繰り返す

2分探索法のアルゴリズム

```
2分探索法(x,s[]){
  left=0; right=n-1;
  do{
    mid = (left+right)/2;
    if x < s[mid] then
      right = mid-1;
    else
      left = mid+1;
  }while(x != s[mid] && left≦right);
  if x == s[mid] then return mid;
  else return -1;
}
```

探索区間を設定

探索区間の中央を分割地点とする

分割地点の値よりも小さい？

区間の右端を分割地点の手前

区間の左端を分割地点の直後

ループを出るのは
- s[mid]と同じとき　または
- 探索区間の幅が0以下

2分探索法の計算時間

- 毎回探索範囲は半分以下になるので,
  $n/2^k = 1$より,
  $k = \log_2 n$
  - n: 探索区間長
  - k: 繰り返し回数

```
left=0; right=n-1;
do{
  mid = (left+right)/2;
  if x < s[mid] then right = mid-1;
  else left = mid+1;
}while(x != s[mid] && left≦right);
if x == s[mid] then return mid;
else return -1;
```

- よって計算時間は O(log n)

2分探索法の変形

- Q. ループ毎のx!=s[mid]の検査は必要？

```
2分探索法(x, s[]){
  left=0; right=n-1;
  do{
    mid = (left+right)/2;
    if x < s[mid] then
      right = mid-1;
    else
      left = mid+1;
  }while(x != s[mid] && left≦right);
  if x == s[mid] then return mid;
  else return -1;
}
```

x < s[mid]
➔ x != s[mid]

2分探索法の変形: x!=s[mid]を省いてみる

- Q. 以下のアルゴリズムは正しい?

NO!

```
2分探索法(x, s[]){
  left=0; right=n-1;
  do{
    mid = (left+right)/2;
    if x < s[mid] then
      right = mid-1;
    else
      left = mid+1;
  }while(left≦right);
  if x == s[mid] then return mid;
  else return -1;
}
```

2分探索法の変形: x!=s[mid]を省いてみる

- Q. 以下のアルゴリズムは正しい?

NO!

```
2分探索法(x, s[]){
  left=0; right=n-1;
  do{
    mid = (left+right)/2;
    if x < s[mid] then
      right = mid-1;
    else
      left = mid+1;
  }while(left≦right);
  if x == s[mid] then return mid;
  else return -1;
}
```

s[k]<=x<s[k+1]となるkが存在したとして脱出条件を分析

- k<midのとき
- mid<=kのとき

ここがまずい

2分探索法の変形: ケース1 . k<midのとき

- k+1 <= mid
  ⇔ /* s[k] <= x < s[k+1] */
  x < s[k+1] <= s[mid]
  ➔ /* def. of algorithm */
  right=mid-1
  ➔ /* k+1 <= mid */
  k <= right

```
left=0; right=n-1;
do{
  mid = (left+right)/2;
  if x < s[mid] then
    right = mid-1;
  else
    left = mid+1;
}while(left≦right);
```

- rightの値は, n-1から出発して単調に減少するが, kより小さくなることはない

2分探索法の変形: ケース2 . mid<=kのとき

- mid <= k
  ⇔ /* s[k] <= x < s[k+1] */
  s[mid] <= s[k] <=x
  ➔ /* def. of algorithm */
  left = mid + 1
  ➔ /* s[mid+1] <= s[k+1] */
  s[left] <= s[k+1]
- leftの値は0から出発して単調に増加するが, k+1より大きくなることはない

```
left=0; right=n-1;
do{
  mid = (left+right)/2;
  if x < s[mid] then
    right = mid-1;
  else
    left = mid+1;
}while(left≦right);
```

2分探索法の変形

- 分かったこと:
  - k right
  - left k+1
- よって，終了条件 left > right を満たすのは left = k+1, right = k のとき
  ➔
  ループから出たときに条件x == s[right]を調べればよい!
  - x==s[right]でなければ配列にxは含まれない

2分探索法の変形: 正しいアルゴリズム……?

例外処理

このようなs[k]または
s[k+1]が存在しないときは？

```
2分探索法(x, s[]){
  left=0; right=n-1;
  do{
    mid = (left+right)/2;
    if x < s[mid] then
      right = mid-1;
    else
      left = mid+1;
  }while(left≦right);
  if x == s[right] then return right;
  else return -1;
}
```

s[k]<=x<s[k+1]となるkが存在したとして脱出条件を分析

- k<midのとき
- mid<=kのとき

2分探索法の変形: 例外の整理

Q. s[k]<=x<s[k+1]不成立はいつ？

- s[k+1]が存在しない
  n<=k+1 ⇔ n-1<=k ⇔ s[n-1] <= x
- s[k]が存在しない
  k+1 == 0 ⇔ x < s[0]

2分探索法の変形: 例外の整理
ケース1. s[n-1]<=x

- 常にs[mid]<=x ➔
  - left = mid+1が繰り返し実行
  - rightの値は変わらない
- ループ脱出時:
  right = n-1

➔ x == s[right]で判定可能

```
2分探索法(x, s[]){
  left=0; right=n-1;
  do{
    mid = (left+right)/2;
    if x < s[mid] then
      right = mid-1;
    else
      left = mid+1;
  }while(left≦right);
  if x == s[right] then
    return mid;
  else return -1;
}
```

2分探索法の変形: 例外の整理
ケース2. x<s[0]

- 常にx < s[mid]➔
  - rightの値だけが減少
- ループ脱出時:
  - right=-1
- s[right]=s[-1]となってしまう！

```
2分探索法(x, s[]){
  left=0; right=n-1;
  do{
    mid = (left+right)/2;
    if x < s[mid] then
      right = mid-1;
    else
      left = mid+1;
  }while(left≦right);
  if x == s[right] then
    return right;
  else return -1;
}
```

2分探索法の変形: 正しいアルゴリズム

```
2分探索法(x, s[]){
  left=0; right=n-1;
  if(x<s[0]) return -1;
  do{
    mid = (left+right)/2;
    if x < s[mid] then
      right = mid-1;
    else
      left = mid+1;
  }while(left≦right);
  if x == s[right] then return right;
  else return -1;
}
```

x<s[0]を特別扱い

Hush? Hash!

ハッシュ法

ハッシュ法

- これまでのデータ管理方法:
  配列に順にデータを並べる

データを順に詰める

- ハッシュ法: データを配列の中に分散

データxをどこに配置する？

ハッシュ関数を用いて決定

データx ⟶ 配列内の位置（インデックス）

データの格納方法

1. データxに対するハッシュ関数の値jを計算
2. 配列のj番目から空いている場所を探し，そこにxを格納
   （同じハッシュ関数値を与えるデータが他にありうる）

```
ハッシュ表htb[0]～htb[m-1]の内容を0に初期化;
for i=0 to n-1 do{
    i番目のデータをxとする;
    j = hash(x);              //ハッシュ関数の値を計算
    while(htb[j] != 0)        //ハッシュ表で空いている場所を探す
        j = (j+1) % m;        //次の場所へ移動
    htb[j] = x;               //最初の空き場所にxを格納
}
```

配列のサイズ（ハッシュテーブルサイズ）をmとする。
0はその場所にまだデータが格納されていないことを示す

例題:

集合S = {3, 4, 6, 7, 9, 11, 14, 15, 17, 18, 20, 23, 24, 26, 27, 29, 30, 32}

ハッシュ関数 hash(x) = x mod 27

（ハッシュ表のサイズは27）

3→3	11→11	20→20	29→2
4→4	14→14	23→23	30→3
6→6	15→15	24→24	32→5
7→7	17→17	26→26	
9→9	18→18	27→ 0	

右がハッシュ関数値

この順に蓄えたとすると，赤字の部分が衝突する.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
htb	27	0	29	3	4	30	6	7	32	9	0	11	0	0

	14	15	16	17	18	19	20	21	22	23	24	25	26
htb	14	15	0	17	18	0	20	0	0	23	24	0	26

ハッシュ法: 検索

- データxに対するハッシュ関数の値jを計算
  - xと等しい要素: 終了
  - xと異なる値が入りかつ0でない: 次の場所を探す
  - 0の場所を参照: xに等しいものは存在しない

```
ハッシュ表の探索(x){
  j = hash(x);
  while( htb[j] != 0 かつ htb[j] != x )
     j = (j+1) % m;      //次の場所へ移動
  if htb[j] == x then jを返して終了;
  else -1を返して終了;
}
```

ハッシュ法: 検索例

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
htb	27	0	29	3	4	30	6	7	32	9	0	11	0	0

	14	15	16	17	18	19	20	21	22	23	24	25	26
htb	14	15	0	17	18	0	20	0	0	23	24	0	26

x=14の場合： hash(14)=14だから，htb[14]から探索し，発見.

x=32の場合： hash(32)=5だから，htb[5]から探索し，30, 6, 7の0でない要素を見た後で32を発見.

x=41の場合： hash(41)=14だから，htb[14]から探索するが，14, 15の後で0を発見する. よってx=41は含まれていない.

ハッシュ法の性能

- ハッシュ表の占有率（または負荷率）
  $\alpha = n/m$ に依存
  - 平均成功探索回数 $\simeq \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$
  - 平均不成功探索回数 $\simeq \frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$

データの個数には関係ないことに注意！
（ハッシュ表のサイズとデータ数の比が問題）

Bachmann–Landau notation

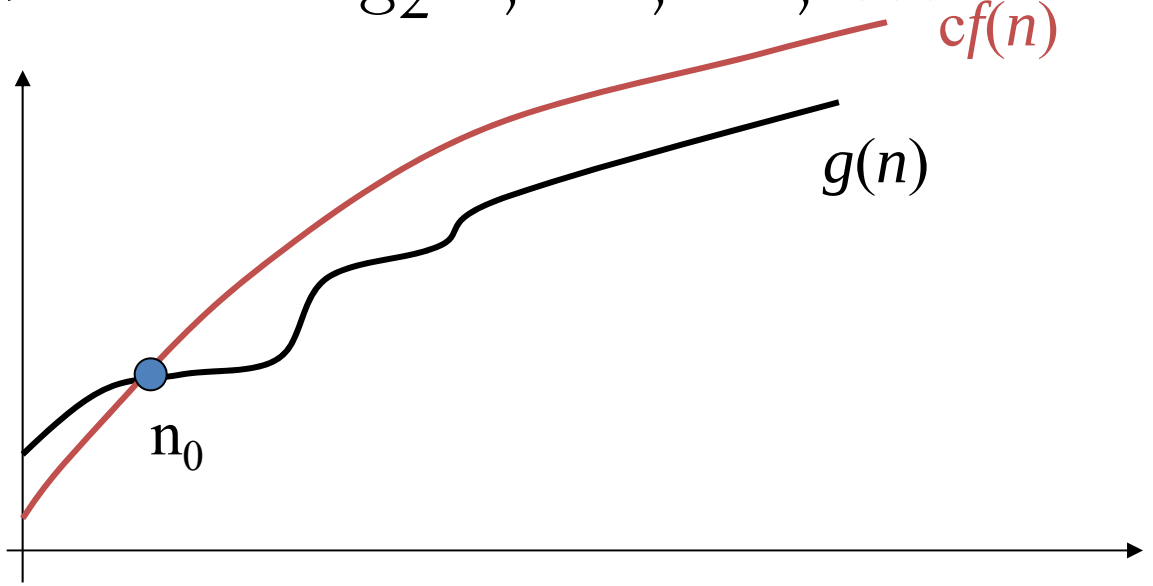
オーダー記法

漸近的計算量

- 入力のサイズnが十分に大きくなったときに計算量がどのような割合で増加するかを表したもの
- 計算量の増加の割合を示すのが目的➔
  - 主要項だけで十分
  - 係数も重要でない.
- 種類:
  - 上界
  - 下界
  - 二つの統合

ビッグオー記法: O(f(n))
計算量の上界を表わす

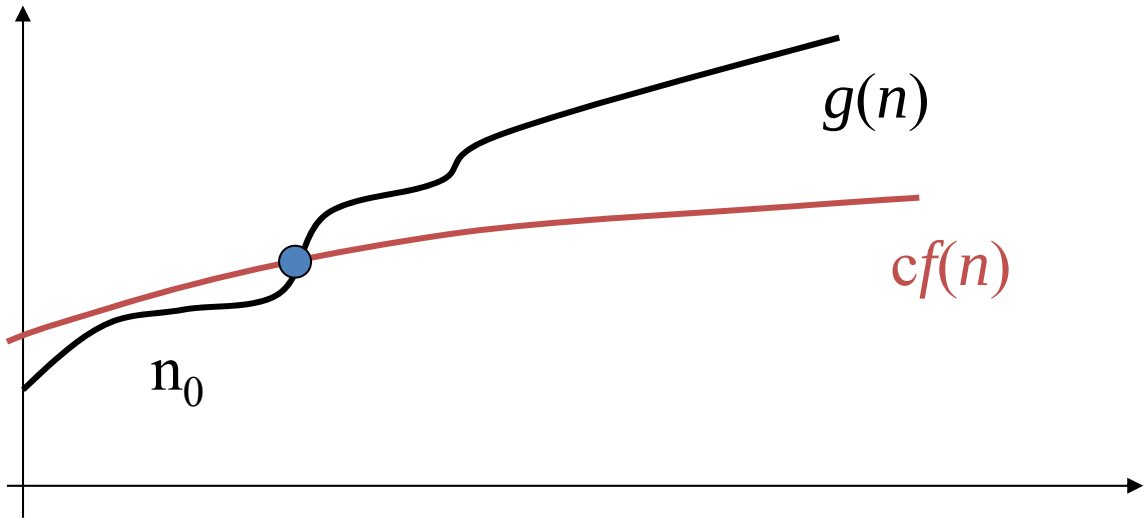
- $O(f(n)) = \{g(n) \mid \exists c > 0, \exists n_0, \forall n \geq n_0, g(n) \leq cf(n)\}$
  - すべての$n \geq n_0$に対して$g(n) \leq cf(n)$であるような正の定数cとn_0が存在する
  - $g(n) \in O(f(n))$でなく$g(n) = O(f(n))$とも書く
- $f(n)$の例: $\log_2 n,\ n^2,\ 2^n,\ \ldots$



$O(\log n)$?

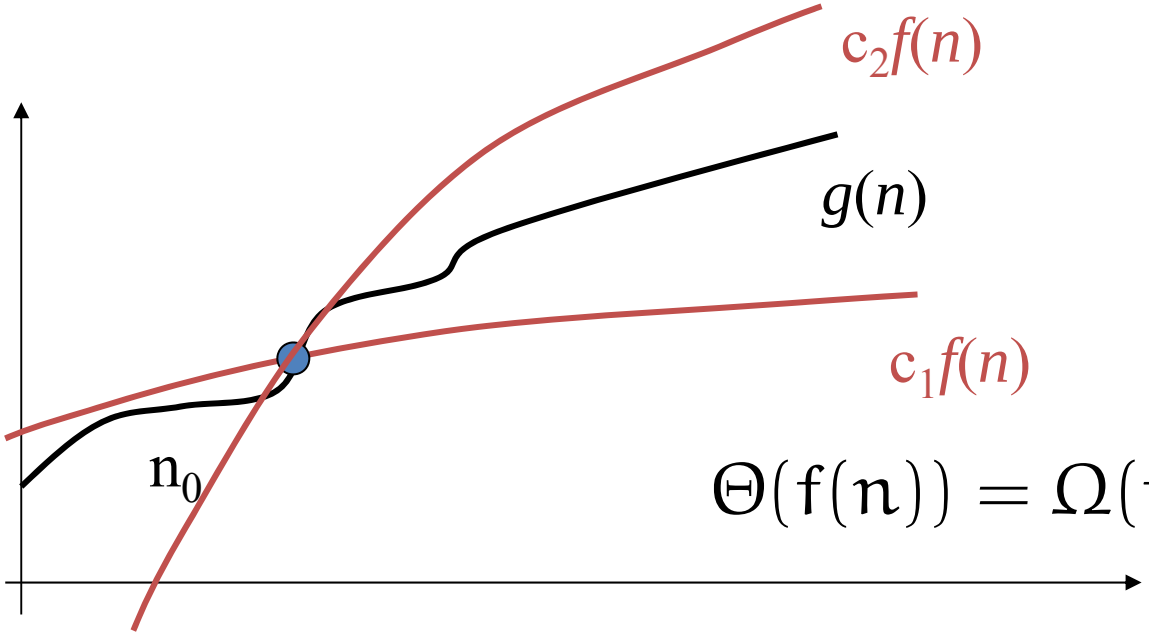
ビッグオメガ記法: Ω(f(n))
計算量の下界を表わす

- $\Omega(f(n)) = \{g(n) \mid \exists c > 0, \exists n_0, \forall n \geq n_0, cf(n) \leq g(n)\}$
  - 全ての $n \geq n_0$ に対して $cf(n) \leq g(n)$ である ような正の定数 c と n_0 が存在する



シータ記法: Θ(f(n))

- $\Theta(f(n)) = \{g(n) \mid \exists c_1, c_2 > 0, \exists n_0, \forall n \geq n_0,$
  $c_1 f(n) \leq g(n) \leq c_2 f(n)\}$
  - 全ての $n \geq n_0$ に対して $c_1 f(n) \leq g(n) \leq c_2 f(n)$ となるような正定数 c_1, c_2, n_0 が存在する



$$\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$$

ミニ演習

- O(n), O(2^n) となるものを選べ
  - 0.1n, $5n^{1000}$, 2.1^n, 2^{n+3}
- $18n^2 + 4n + 2014 \in O(n^2)$ を証明せよ
- O(f(n)) の定義:

$$O(f(n)) = \{g(n) \mid \exists c > 0, \exists n_0, \forall n \geq n_0, g(n) \leq cf(n)\}$$