アルゴリズム論
Theory of Algorithms

第１回講義
アルゴリズムの設計と解析の基礎(1)

アルゴリズム論
Theory of Algorithms

Lecture #1

Foundation of Design and Analysis of Algorithms(1)

アルゴリズム(algorithm)

＝ 問題を正しく解くための計算の手順
・どんな入力に対しても正しく解が得られること
・必ず終了すること
・記述に曖昧さがないこと

プログラム(program)
＝ アルゴリズムを計算機言語で記述したもの
あるいは、単なる命令の系列

思い付きのアルゴリズム
:アルゴリズム設計技法に関する知識の欠如
作りっぱなしのアルゴリズム
:アルゴリズムの動作の解析がない
・計算時間を推定する式
最も都合のよい場合、都合の悪い場合
・必要なメモリー量を推定する式
・アルゴリズムの正しさの検証

Algorithm

＝procedure to solve a problem correctly

- to find a correct solution for any input
- to terminate in all cases
- no ambiguity in its description

Program

＝ description of algorithms in computer languages
or simply a sequence of instructions

Algorithms based on sudden thought

：lack of knowledge of algorithm design schema

Algorithms without any consideration

：no analysis on behavior of algorithm

- equation to estimate computation time
  best case, worst case
- equation to estimate storage required
- validation of correctness of algorithms

最小値

問題P0：配列に蓄えられたn個のデータの最小値を求めよ.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31

アルゴリズムP0-A0:

```
min=9999;
for( i=0; i<n; i++ )
   if( a[i] < min ) min = a[i];
return min;
```

データ比較回数はn回. 計算時間はO(n).

すべてのデータが9999以下なら，正しく最小値が求まる.
10000以上の値が含まれると9999が出力される.

Minimum Value

Problem P0：Find a minimum value among n data in an array.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31

Algorithm P0-A0:

```
min=9999;
for( i=0; i<n; i++ )
  if( a[i] < min ) min = a[i];
return min;
```

number of comparisons is n. computation time is O(n).

If all data are at most 9999, then the minimum value is found correctly, but 9999 is output otherwise.

最小値

問題P0：配列に蓄えられたnつのデータの最小値を求めよ.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31

アルゴリズムP0-A1:

```
min=a[0];
for( i=1; i<n; i++ )
   if( a[i] < min ) min = a[i];
return min;
```

データ比較回数はn-1回. 計算時間はO(n).

常に正しく最小値を求める.

Minimum value

Problem P0：Find a minimum value among n data in an array.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31

Algorithm P0-A1:

```
min=a[0];
for( i=1; i<n; i++ )
  if( a[i] < min ) min = a[i];
return min;
```

number of data comparisons is n-1. computation time is O(n).
Minimum value is always found correctly.

再帰を用いた方法

問題P0：配列に蓄えられたn個のデータの最小値を求めよ．

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31

Min(i) = a[0] - a[i]の最小値，と定義すると，

Min(0) = a[0];

Min(i) = min(Min(i-1), a[i])   for i>0

これをプログラムに直すと

アルゴリズムP0-A2:

```
int Min(int i){
  if(i==0) return a[0];
  else if(a[i] < Min(i-1) ) return a[i];
  else return Min(i-1);
}
main で cout << Min(n-1)  とする.
```

計算時間は？

Min(i-1)を2回呼び出すと効率が悪い

解析は？

Algorithms based on Recursion

Problem P0：Find a minimum value among n data in an array.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31

Define Min(i) = minimum among a[0] - a[i], then

Min(0) = a[0];

Min(i) = min(Min(i-1), a[i])   for i>0

Converting the above into a program:

Algorithm P0-A2:

```
int Min(int i){
  if(i==0) return a[0];
  else if(a[i] < Min(i-1) ) return a[i];
  else return Min(i-1);
}
main で cout << Min(n-1)  とする.
```

Computation time?

If Min(i-1) is called twice, then it is not efficient.

Analysis?

アルゴリズムP0-A2:

```
int Min(int i){
  if(i==0) return a[0];
  else if(a[i] < Min(i-1) ) return a[i];
  else return Min(i-1);
}
```

main で cout << Min(n-1)　とする.

練習問題:アルゴリズムP0-A2を実装し，動作を確かめよ.

計算時間をT(n)と表すと,

$T(n) \leqq 2T(n-1)+c$

cは定数.

$T(n) \leqq 2T(n-1)+c \leqq 2(2T(n-2)+c)+c=2^2T(n-2)+(2+1)c$

$\leqq 2^{n-1}T(n-(n-1))+(2^{n-2}+...+2+1)c = O(2^n)$

となり，指数時間かかってしまう危険性がある.

練習問題: アルゴリズムP0-A2が指数時間かかってしまうような入力を具体的に与えよ.

Algorithm P0-A2:

```
int Min(int i){
  if(i==0) return a[0];
  else if(a[i] < Min(i-1) ) return a[i];
  else return Min(i-1);
}
main contains cout << Min(n-1).
```

Exercise：Implement the algorithm P0-A2 to see its behavior.

If we denote the computation time by T(n), then we habe

$T(n) \leqq 2T(n-1)+c$

where c is a constant.

$T(n) \leqq 2T(n-1)+c \leqq 2(2T(n-2)+c)+c=2^2T(n-2)+(2+1)c$

$\leqq 2^{n-1}T(n-(n-1))+(2^{n-2}+...+2+1)c = O(2^n)$

This suggest some possibility of exponential time.

Exercise: Give an input such that the algorithm P0-A2 requires exponential time.

では，どうすれば指数時間を回避できるか？

同じ関数を重複して呼び出さないように注意.
Min(i-1)の値を変数に蓄えておく.

アルゴリズムP0-A3:

```
int Min(int i){
  if(i==0) return a[0];
  minsf = Min(i-1);
  if(a[i] < minsf ) return a[i];
  else return minsf;
}
```

main で cout << Min(n-1) とする.

計算時間の解析

$T(n) \leqq T(n-1) + c.$

したがって，$T(n) = O(n)$.

他にも再帰的なアルゴリズムは考えられるか？

Then, how can we avoid exponential time?

The same function should be never called twice.

Store the value of Min(i-1) in a variable.

Algorithm P0-A3:

```
int Min(int i){
  if(i==0) return a[0];
  minsf = Min(i-1);
  if(a[i] < minsf ) return a[i];
  else return minsf;
}
main contains  cout << Min(n-1).
```

Computation time

$T(n) \leqq T(n-1) + c$.

Thus, $T(n) = O(n)$.

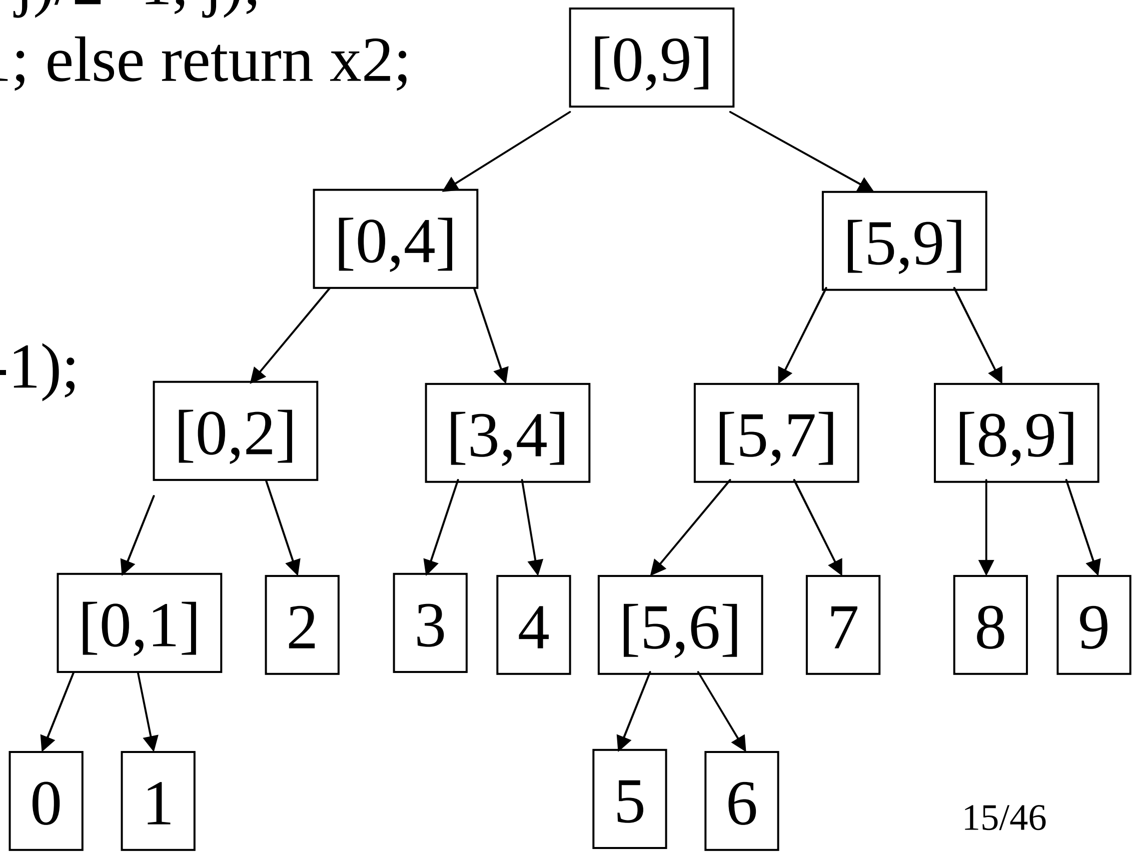
Any other recursive algorithm?

分割統治法

アルゴリズムP0-A4:

```
int find_min(int i, int j){
  if(i==j) return a[i];
  int x1 = find_min(i, (i+j)/2);
  int x2 = find_min((i+j)/2+1, j);
  if( x1 < x2) return x1; else return x2;
}
main(){
...
cout << find_min(0, n-1);
...
}
```

与えられた配列を前半と後半に2分割し，それぞれで再帰的に最小値を求め得られた2つの最小値の小さい方を答える．



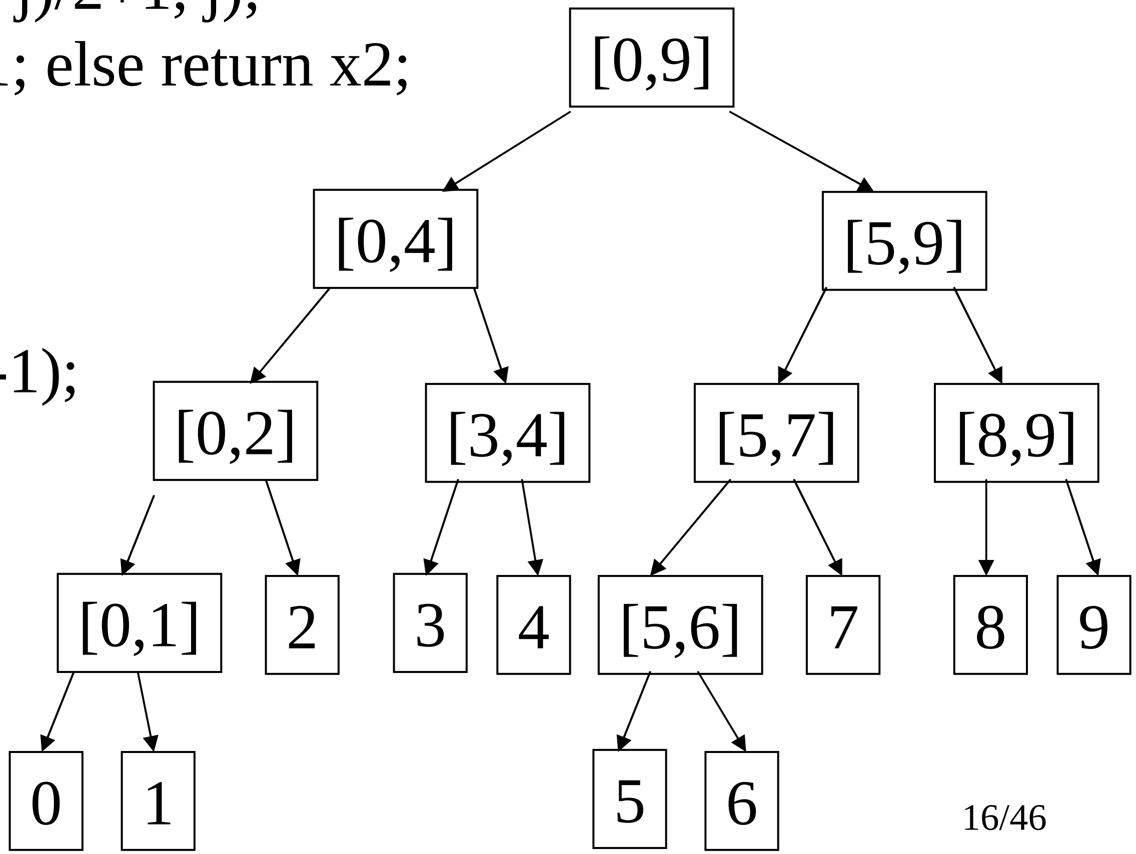
練習問題：データ比較回数を求めよ．

Divide-and-Conquer

Algorithm P0-A4:

```
int find_min(int i, int j){
    if(i==j) return a[i];
    int x1 = find_min(i, (i+j)/2);
    int x2 = find_min((i+j)/2+1, j);
    if( x1 < x2) return x1; else return x2;
}
main(){
...
cout << find_min(0, n-1);
...
}
```

Divide an array into two halves, find minimum values recursively, and output the smaller one of the two.



Exercise: Analyze the number of comparisons.

問題P1：配列に蓄えられたn個のデータそれぞれについて，自分より左（自分も含めて）の要素の中の最小値を求めよ.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31
	17	17	17	17	17	16	16	16	16	16

アルゴリズムP1-A0:

```
lmin[0] = a[0];
for( i=1; i<n; i++ ) {
   min=a[0];
   for(j=1; j<=i; j++)
      if( a[j] < min ) min = a[j];
   lmin[i] = min;
}
```

腕力法：
すべての要素について問題P0に対するアルゴリズムを適用.

計算時間は明らかに
$O(n^2)$

Problem P1: For each datum from n data in an array find the minimum value among those to its left (including itself).

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31
	17	17	17	17	17	16	16	16	16	16

Algorithm P1-A0:

```
  lmin[0] = a[0];
  for( i=1; i<n; i++ ) {
    min=a[0];
    for(j=1; j<=i; j++)
      if( a[j] < min ) min = a[j];
    lmin[i] = min;
  }
```

Brute-Force algorithm:
Apply the algorithm for the problem P0 for each element.

Computation time is obviously $O(n^2)$

問題P1：配列に蓄えられたn個のデータそれぞれについて，自分より左（自分も含めて）の要素の中の最小値を求めよ.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31
	17	17	17	17	17	16	16	16	16	16

lmin[i-1]　a[i]

lmin[i] = min(lmin[i-1], a[i])であることに注目すると

アルゴリズムP1-A1:

```
  lmin[0] = a[0];
  for( i=1; i<n; i++ ){
    min=lmin[i-1];
    if(a[i] < min) min = a[i];
    lmin[i] = min;
  }
```

計算時間は
O(n)

Problem P1：For each datum from n data in an array find the minimum value among those to its left (including itself).

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31
	17	17	17	17	17	16	16	16	16	16

lmin[i-1]   a[i]

If we note that lmin[i] = min(lmin[i-1], a[i])

Algorithm P1-A1:

```
lmin[0] = a[0];
for( i=1; i<n; i++ ){
  min=lmin[i-1];
  if(a[i] < min) min = a[i];
  lmin[i] = min;
}
```

Computation time is O(n)

問題P2：n個のデータが配列a[]に蓄えられているとき，区間[p,q]，$0 \leqq p<q<n$,に対して定まる差（区間差）a[q] - a[p] を最大にする区間[p, q]を求めよ.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31

a[p] a[q]

すべての区間を列挙して，最大の区間差を求めればよい.

アルゴリズムP2-A0:

```
maxsf=0;
for( p=0; p<n-1; p++ )
   for(q=p+1; q<n; q++)
      if(a[q] - a[p] > maxsf) maxsf = a[q] - a[p];
```

２重ループの構造なので，計算時間は$O(n^2)$

Problem P2 : When n data are stored in an array a[], find an interval to maximize an interval difference a[q]-a[p] for an interval [p,q], $0 \leqq p<q<n$.

	0	1	2	3	4	5	6	7	8	9
a	17	32	19	22	28	16	18	20	39	31

a[p] a[q]

Find the largest interval difference by enumerating all intervals.

Algorithm P2-A0:

```
  maxsf=0;
  for( p=0; p<n-1; p++ )
    for(q=p+1; q<n; q++)
      if(a[q] - a[p] > maxsf) maxsf = a[q] - a[p];
```

Double loop structure ==> computation time is $O(n^2)$.

pとqの順序を入れ替えてループを構成してみると，

```
1:   maxsf=0;
2:   for( q=1; q<n; q++ )
3:       for(p=0; p<q; p++)
4:           if(a[q] - a[p] > maxsf)
5:               maxsf = a[q] - a[p];
```

3-5行目では，a[0]～a[q-1]の最小値を求めている．
よって，問題P1のように先に各要素について自分より左での最小値をO(n)時間で求めておけば，この部分を簡単化可能．

アルゴリズムP2-A1:
アルゴリズムP1-A1でa[0]～a[q-1]の最小値をlmin[q-1]として求めておく．

```
maxsf=0;
  for( q=1; q<n; q++ )
        if(a[q] - lmin[q-1] > maxsf)
            maxsf = a[q] - lmin[q-1];
```

最初のステップはO(n).
残りの計算もO(n).
よって，全体でもO(n).

余分の配列lmin[]を使わずに同じことができるか？

Reconstructing the program by exchanging the order of p and q

```
1:   maxsf=0;
2:   for( q=1; q<n; q++ )
3:       for(p=0; p<q; p++)
4:           if(a[q] - a[p] > maxsf)
5:               maxsf = a[q] - a[p];
```

The lines3-5 find the minimum value of a[0]～a[q-1]. Thus, this part can be simplified if for each element the minimum value to its left is available as in Problem 1.

Algorithm P2-A1:
Find the minimum value among a[0]～a[q-1] as lmin[q-1] by the algorithm P1-A1.

```
maxsf=0;
  for( q=1; q<n; q++ )
      if(a[q] - lmin[q-1] > maxsf)
          maxsf = a[q] - lmin[q-1];
```

The first step takes O(n) time. The computation of the remaining steps is also O(n). Thus, the total computation time is O(n).

Is it possible without any auxiliary array lmin[]?

```
アルゴリズムP1-A1:
  lmin[0] = a[0];
  for( q=1; q<n; q++ )
    min=lmin[q-1];
    if(a[q] < min) min = a[q];
    lmin[q] = min;
  }
```

```
maxsf=0;
  for( q=1; q<n; q++ )
      if(a[q] - lmin[q-1] > maxsf)
        maxsf = a[q] - lmin[q-1];
```

これらを組み合わせると次のアルゴリズムを得る.

```
アルゴリズムP2-A2:
  maxsf=0;min=a[0];
  for( q=1; q<n; q++ ){
     if(a[q] - min > maxsf) maxsf = a[q] - min;
     if(a[q] < min) min = a[q];
  }
```

計算時間:
１重ループだから
$O(n)$

```
Algorithm P1-A1:
  lmin[0] = a[0];
  for( q=1; q<n; q++ )
    min=lmin[q-1];
    if(a[q] < min) min = a[q];
    lmin[q] = min;
  }
```

```
maxsf=0;
  for( q=1; q<n; q++ )
      if(a[q] - lmin[q-1] > maxsf)
        maxsf = a[q] - lmin[q-1];
```

Combination of the two algorithm leads to the following algorithm.

```
Algorithm P2-A2:
  maxsf=0;min=a[0];
  for( q=1; q<n; q++ ){
     if(a[q] - min > maxsf) maxsf = a[q] - min;
     if(a[q] < min) min = a[q];
  }
```

Computation time
  Single loop
  => O(n)

問題P3（最大区間和）: n個のデータが配列a[]に蓄えられているとき,区間[p,q]に対する和（区間和）sum(p, q)を，その区間内の要素a[p]～a[q]の和と定義する．このとき，区間和の最大値を求めよ．

	0	1	2	3	4	5	6	7	8	9
a	10	-9	-5	12	-3	10	-8	11	-8	-2

p \ q	0	1	2	3	4	5	6	7	8	9
0	10	1	-4	8	5	15	7	18	10	8
1		-9	-14	-2	-5	5	-3	8	0	-2
2			-5	7	4	14	6	17	9	7
3				12	9	19	11	**22**	14	12
4					-3	7	-1	10	2	0
5						10	2	13	5	3
6							-8	3	-5	-7
7								11	3	1
8									-8	-10
9										-2

Problem P3（Largest Sum Interval）: Given n data in an array a[], a sum interval sum(p,q) for an interval [p,q] is defined as the sum of elements a[p]～a[q]. Find a largest sum interval for a given array.

	0	1	2	3	4	5	6	7	8	9
a	10	-9	-5	12	-3	10	-8	11	-8	-2

p \ q	0	1	2	3	4	5	6	7	8	9
0	10	1	-4	8	5	15	7	18	10	8
1		-9	-14	-2	-5	5	-3	8	0	-2
2			-5	7	4	14	6	17	9	7
3				12	9	19	11	**22**	14	12
4					-3	7	-1	10	2	0
5						10	2	13	5	3
6							-8	3	-5	-7
7								11	3	1
8									-8	-10
9										-2

問題P3（最大区間和）: n個のデータが配列a[]に蓄えられているとき,区間[p,q]に対する和（区間和）sum(p, q)を，その区間内の要素a[p]～a[q]の和と定義する．このとき，区間和の最大値を求めよ．

すべての区間について対応する区間和を求めればよい．

アルゴリズムP3-A0:

```
maxsum=0;
for(p=0; p<n; p++)
  for(q=p; q<n; q++){
    // 区間[p,q]での和sumを求める
    sum=0;
    for(i=p; i<=q; i++)
      sum = sum + a[i];
    if(sum > maxsum) maxsum = sum;
  }
```

計算時間:

３重ループだから

O(n^3)時間

Problem P3（Largest Sum Interval）: Given n data in an array a[], a sum interval sum(p,q) for an interval [p,q] is defined as the sum of elements a[p]～a[q]. Find a largest sum interval for a given array.

It can be computed by computing the interval sum for every interval.

Algorithm P3-A0:

```
maxsum=0;
for(p=0; p<n; p++)
  for(q=p; q<n; q++){
    // find the interval sum in an interval [p,q]
    sum=0;
    for(i=p; i<=q; i++)
      sum = sum + a[i];
    if(sum > maxsum) maxsum = sum;
  }
```

Computation time:
triple-loop=>
$O(n^3)$ time

詳細な解析

$$\sum_{p=0}^{n-1}\sum_{q=p}^{n-1}\sum_{i=p}^{q} c = \sum_{p=0}^{n-1}\sum_{q=p}^{n-1}(q-p+1)c$$

$$= \sum_{p=0}^{n-1} c((1/2)n(n-1)-p(p-1))/2-(p-1)(n-p)c=O(n^3)$$

（改良）

区間の左端pを固定して考えると,

右端qは一つずつ右へ移動する.

区間和の変化は右端の要素a[q]の分だけ.

　　これはO(1)時間で更新可能.

Detailed analysis

$$\sum_{p=0}^{n-1}\sum_{q=p}^{n-1}\sum_{i=p}^{q} c = \sum_{p=0}^{n-1}\sum_{q=p}^{n-1}(q-p+1)c$$
$$= \sum_{p=0}^{n-1} c((1/2)n(n-1)-p(p-1))/2-(p-1)(n-p)c=O(n^3)$$

（Improvement）

If we fix the left endpoint p of an interval,
the right endpoint moves tot he right one by one.
The interval sum is affected only by the rightmost
element a[q].
This update is maintained in O(1) time.

繰り返し計算での重複を排除すると

アルゴリズムP3-A1:

```
maxsum=a[0];
for(p=0; p<n; p++){
   sum=0;
   for(q=p; q<n; q++){
      sum = sum + a[q];
      if( sum > maxsum) maxsum = sum;
   }
}
return maxsum;
```

２重ループの構造になったので，計算時間は $O(n^2)$

冗長性:

和の計算で同じ区間の和が何度も計算されている.

繰り返し計算での重複を排除すると効率が改善される.

Removing duplication in the iteration, we have

```
Algorithm P3-A1:
maxsum=a[0];
for(p=0; p<n; p++){
   sum=0;
   for(q=p; q<n; q++){
      sum = sum + a[q];
      if( sum > maxsum) maxsum = sum;
   }
}
return maxsum;
```

double-loop structure
=> $O(n^2)$ time

Redundancy:

The same interval is dealt with in the computation of sums more than once. Thus, if we remove duplication in the iteration then the efficiency is improved.

最大区間和を与える区間の満たすべき性質

左端の要素a[p]は全体の平均値より大きいこと.

=>"極大区間"の概念へと発展可能.

例：$\sum_{i=0}^{n-1} a_i = 0$ とする

アルゴリズムP3-A2:

```
a[0]～a[n-1]の平均値averageを求める.
maxsum=a[0];
for(p=0; p<n-1; p++){
  if( a[p]>average){
    sum=0;
    for(q=p; q<n; q++){
      sum = sum + a[q];
      if( sum > maxsum) maxsum = sum;
    }
  }
return maxsum;
```

平均値以上の要素数はn/2個以上になり得るので，結局計算時間は$O(n^2)$.

Property to be satisfied by the largest sum interval

the leftmost element a[p] must be larger than the overall average

=> leads to a notion of "maximal interval"

Ex: Suppose $\sum_{i=0}^{n-1} a_i = 0.$

Algorithm P3-A2:

```
Find the `average` of a[0]∼a[n-1].
maxsum=a[0];
for(p=0; p<n-1; p++){
  if( a[p]>average){
    sum=0;
    for(q=p; q<n; q++){
      sum = sum + a[q];
      if( sum > maxsum) maxsum = sum;
    }
  }
return maxsum;
```

Since the number of elements can be more than n/2, the total computation time is $O(n^2)$.

全く別の考え方によるアルゴリズム

S[i] = a[0]〜a[i]の和，と定義すると，区間[p,q]の和は

sum(p,q) = sum(0,q) - sum(0,p-1) =S[q] - S[p-1]

として計算できる．したがって，S[0], S[1], ... , S[n-1]を求めておけば，区間差の最大値を求める問題と等しくなる．

アルゴリズムP3-A3:

```
  S[0] = a[0];
  for(i=1; i<n; i++)
     S[i] = S[i-1] + a[i];
  maxsum=a[0]; minsf=a[0];
  for(p=1; p<n; p++){
      if(S[p] - minsf > maxsum) maxsum = S[p] - minsf;
      if(S[p] < minsf) minsf = S[p];
   }
   return maxsum;
```

計算時間

O(n)

Algorithm based on completely different ideas

If we define S[i] = sum of a[0]～a[i], the interval sum for [p,q] can be computed by

sum(p,q) = sum(0,q) - sum(0,p-1) =S[q] - S[p-1].

Thus, if we have S[0], S[1], ... , S[n-1] in advance then the problem is reduced to that of finding the largest interval difference.

Algorithm P3-A3:

```
S[0] = a[0];
for(i=1; i<n; i++)
  S[i] = S[i-1] + a[i];
maxsum=a[0]; minsf=a[0];
for(p=1; p<n; p++){
   if(S[p] - minsf > maxsum) maxsum = S[p] - minsf;
   if(S[p] < minsf) minsf = S[p];
 }
 return maxsum;
```

Computation time
O(n)

作業用配列なしでも可能か？

ループの中では配列S[]に関してはS[i]の値しか参照していない.
=>和を配列で管理する必要はない.
S[i]を求めるループと区間和最大値を求めるループをまとめる.

アルゴリズムP3-A4:

```
maxsum=a[0]; minsf=a[0];sum=a[0];
for(p=1; p<n; p++){
    sum = sum + a[p];
    if(sum - minsf > maxsum) maxsum = sum - minsf;
    if(sum < minsf) minsf = sum;
 }
 return maxsum;
```

計算時間はやはりO(n).

Is it possible without auxiliary array?

In the loop we refer only S[i] in the array S[].
=>no need to maintain sums in an array
Combining the loop to find S[i] and that of finding the largest sum interval, we have

Algorithm P3-A4:

```
maxsum=a[0]; minsf=a[0];sum=a[0];
for(p=1; p<n; p++){
    sum = sum + a[p];
    if(sum - minsf > maxsum) maxsum = sum - minsf;
    if(sum < minsf) minsf = sum;
}
return maxsum;
```

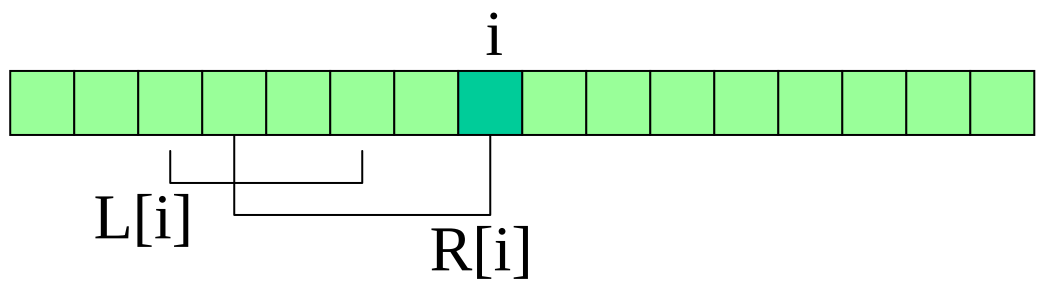
Computation time is still O(n).

動的計画法に基づくアルゴリズム

配列を左から右に順に調べていく.
a[i]を調べているとき, [0,i-1]の範囲における最大の区間和をL[i],
a[i]を右端とする区間の中での最大区間和をR[i]とする.
このとき,

$$L[i] = \begin{cases} L[i-1] & L[i-1] \geqq R[i-1] \text{のとき,} \\ R[i-1] & \text{それ以外のとき.} \end{cases}$$

$$R[i] = \begin{cases} a[i] & R[i-1]+a[i] < a[i] \text{のとき,} \\ R[i-1] + a[i] & \text{それ以外のとき.} \end{cases}$$

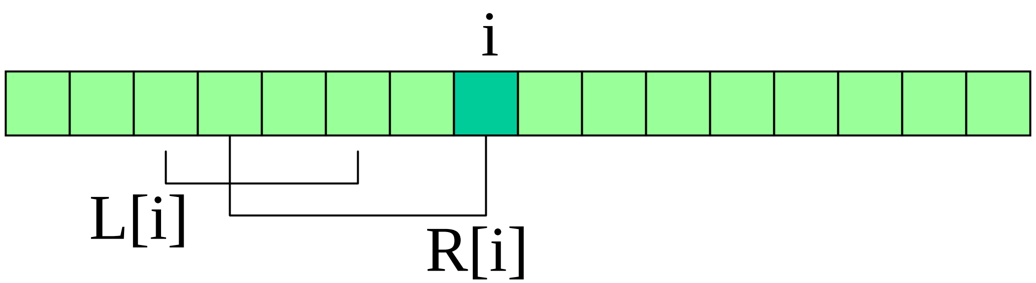


Algorithm based on dynamic programming

The array is checked from left to right.
Let L[i] be the largest sum interval in the interval [0, i-1] and
R[i] be the largest sum interval for interval with a[i] in its right end.
Then, we have

$$L[i] = \begin{cases} L[i-1] & \text{if } L[i-1] \geqq R[i-1], \\ R[i-1] & \text{otherwise.} \end{cases}$$

$$R[i] = \begin{cases} a[i] & \text{if } R[i-1]+a[i]<a[i], \\ R[i-1] + a[i] & \text{otherwise.} \end{cases}$$



$$L[i] = \begin{cases} L[i-1] & L[i-1] \geqq R[i-1]\text{のとき}, \\ R[i-1] & \text{それ以外のとき}. \end{cases}$$

$$R[i] = \begin{cases} a[i] & R[i-1]+a[i]<a[i]\text{のとき}, \\ R[i-1] + a[i] & \text{それ以外のとき}. \end{cases}$$

最後にL[n-1]とR[n-1]の大きい方が最大値.

アルゴリズムP3-A5:

```
L[0] = R[0] = a[0];
for(i=1; i<n; i++){
  if( L[i-1] >= R[i-1] ) L[i] = L[i-1]; else L[i] = R[i-1];
  if( R[i-1] + a[i] < a[i] ) R[i] = a[i]; else R[i] = R[i-1] + a[i];
}
if( L[n-1] > R[n-1] ) return L[n-1]; else return R[n-1];
```

計算時間はO(n).

作業用の配列をなくす事はできるか？

$$L[i] = \begin{cases} L[i-1] & \text{if } L[i-1] \geqq R[i-1], \\ R[i-1] & \text{otherwise.} \end{cases}$$

$$R[i] = \begin{cases} a[i] & \text{if } R[i-1]+a[i]<a[i], \\ R[i-1] + a[i] & \text{otherwise.} \end{cases}$$

Finally, we take the larger of L[n-1] and R[n-1] as the maximum.

```
Algorithm P3-A5:
L[0] = R[0] = a[0];
for(i=1; i<n; i++){
  if( L[i-1] >= R[i-1] ) L[i] = L[i-1]; else L[i] = R[i-1];
  if( R[i-1] + a[i] < a[i] ) R[i] = a[i]; else R[i] = R[i-1] + a[i];
}
if( L[n-1] > R[n-1] ) return L[n-1]; else return R[n-1];
```

Computation time is O(n).

Is it possible to do without any auxiliary array?

L[i]の値はL[i-1]とR[i-1]だけで決まる.
R[i]の値はR[i-1], a[i]だけで決まる.
よって，配列を使う必要はない.

アルゴリズムP3-A6:

```
L = R = a[0];
for(i=1; i<n; i++){
  if( L >= R ) L = L; else L = R;
  if( R + a[i] < a[i] ) R = a[i]; else R = R + a[i];
}
if( L > R) return L; else return R;
```

L[i] is determined only by L[i-1] and R[i-1].
R[i] is determined only by R[i-1] and a[i].
Therefore, no auxiliary array is required.

Algorithm P3-A6:

```
L = R = a[0];
for(i=1; i<n; i++){
  if( L >= R ) L = L; else L = R;
  if( R + a[i] < a[i] ) R = a[i]; else R = R + a[i];
}
if( L > R) return L; else return R;
```