2024 機械情報夏学期
ソフトウェア第二

担当：岡田 慧 (k-okada-soft2@jsk.t.u-tokyo.ac.jp)

7. アルゴリズムとデータ構造１（リストとツリー）

1　アルゴリズムとデータ構造

そもそも,「アルゴリズムとデータ構造」というのは，Pascal, Modula-2 を開発したニクラウス・ヴィルト博士 (Niklaus Wirth) による,「アルゴリズム＋データ構造＝プログラム」(1975 年) という本の題名がオリジナルであり，これがソフトウェアの基本構造であるためその後も数多くの同名の教科書が出版されている[1].

また，伽藍とバザール（エリック・レイモンド http://cruel.org/freeware/cathedral.html）の中で,

> これと、その前の変更は、プログラマとして頭にいれておくといい一般原則を示すものだ。特に、ダイナミックなタイプ処理 をしない C みたいな言語では：
>
> 9．賢いデータ構造と間抜けなコードのほうが、その逆よりずっとまし。
>
> またもやフレッド・ブルックス本の第 11 章から。「コードだけ見せてくれてデータ構造は見せてもらえなかったら、わたしはわけがわからぬままだろう。データ構造さえ見せてもらえれば、コードのほうはたぶんいらない。見るまでもなく明らかだから」
>
> ほんとはかれが言ったのは「フローチャート」に「テーブル」だった。でも 30 年にわたる用語面・文化面での推移を考慮すれば、ほとんど同じことを言ってる。

として言及されている．フレッド・ブルックスの本，というのは『人月の神話』と題するものであり，ここでは「遅れているソフトウェア・プロジェクトに人員を投入しても、そのプロジェクトをさらに遅らせるだけである」と主張しており，ソフトウェア工学の分野ではブルックスの法則と呼ばれている．

2　リスト (List) と探索 (Search)・ソート (Sort) アルゴリズム

まずは基本的なデータ構造としてリストを見ていこう．

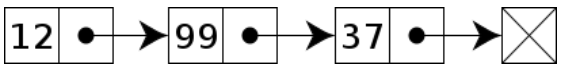
[1]知らない言語について，どのようなものか知りたければ，それぞれの言語で hello world プログラムを書けばよいかを調べるのがよい．http://www.roesler-ac.de/wolfram/hello.htm や WikiPedia の Hello world プログラムの一覧で Psacal, Modula-2 を調べてみよう．

2.1 連結リスト (Linked List)

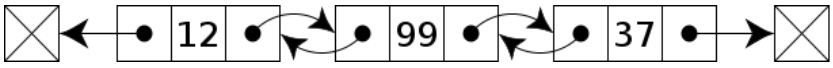
連結リストはの其々のノード (Node) が，別のノードを 1 つだけ指し示しているようなデータの集合である．複数のノードを指し示す場合はグラフ (Graph) または木 (Tree) と呼ぶ．

プログラム上はこれはは配列やリストで実装できるように見えるが，これらが連続したメモリ上のデータとして表されることに対して，連結リストは其々のノードが別の場所に存在してもよく，各ノードには，そのノードの情報を保持するデータ領域の他に，次のノードを指し示す参照を保持する領域を持っている．

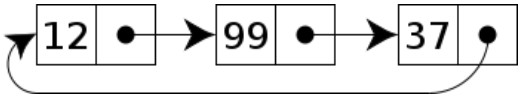
このことを念頭に以下の片方向リスト (Singly Linked List) と双方向リスト (Doubly Linked List) を循環リスト (Circular Linked List) の図を見てほしい．四角の表記は空のリストを表す．また，先頭のノードは Head と呼ばれる．



3 つの整数値を格納した片方向リスト

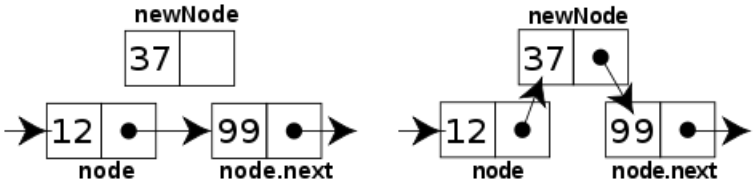


3 つの整数値を格納した双方向リスト

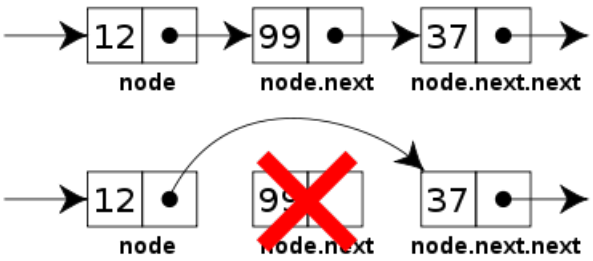


3 つの整数値を格納した循環リスト
片方向リスト，双方向リスト，循環リスト
(http://ja.wikipedia.org/wiki/連結リスト)

連結リストにおいて要素の挿入をする場合は以下のように，挿入したい箇所の前のノードが保持する次のノードの参照先を，新しいノードに変更し，新しいノードの参照先を，挿入したい箇所の次のノードを指すようにする．



また，要素を削除したい場合，削除したいノードの前のノードが保持する次のノードの参照先を，削除したい箇所の次のノードを指すように変更する．



C 言語で連結リストを記述した例を以下に示す.

```
#include <stdio.h>

typedef struct node {
  int data;
  struct node* next;
} node;

void print_list(node *head) {
  for(node* p = head->next;
      p != NULL;
      p = p->next) {
    printf("%d -> ", p->data);
  }
  printf("NULL\n");
}

void insert_after(node *head,
                  node *target,
                  node *new_node) {
  for(node* p = head;
      p != NULL;
      p = p->next) {
    if ( p == target ) {
      new_node->next = p->next;
      p->next = new_node;
    }
  }
}
```

```
void remove_node(node *head, node *target) {
  for(node *p0 = head, *p1 = head->next;
      p1 != NULL;
      p0 = p1, p1 = p1->next) {
    if ( p1 == target ) {
      p0->next = p1->next;
    }
  }
}

int main() {
  node head;
  node node1, node2, node3;
  head.next = &node1;
  node1.data = 12;
  node1.next = &node2;
  node2.data = 99;
  node2.next = NULL;
  node3.data = 37;

  print_list(&head);          // print list

  // insert node3(38)
  insert_after(&head, &node1, &node3);
  print_list(&head);          // print list

  remove_node(&head, &node2); // rmeove node2(99)
  print_list(&head);         // print list
}
```

また, C++言語で連結リストを表すクラスて記述した例を以下に示す.

```
class LinkedList;
class Node {
public:
  Node() {
    this->next = NULL;
  }
  Node(int data) {
    this->data = data;
    this->next = NULL;
  }
  Node(int data, Node *next) {
    this->data = data;
    this->next = next;
  }
  int getData() {
    return this->data;
  }
  void setData(int data) {
    this->data = data;
  }
  Node* getNext() {
    return this->next;
  }
  void setNext(Node *node) {
    this->next = node;
  }
  friend std::ostream&
      operator<<(std::ostream& os,
                 const Node& n);
private:
  int data;
  Node* next;
};

std::ostream&
    operator<<(std::ostream &os,
               const Node& n) {
  os << n.data;
}
```

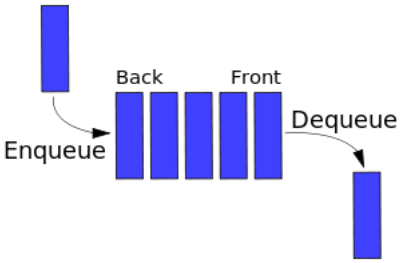
```
class LinkedList {
public:
  LinkedList() {}
  void insertNode(Node& target, Node *new_node) {
    for(Node* p = &head;
        p != NULL;
        p = p->getNext()) {
      if ( p == &target) {
        new_node->setNext(p->getNext());
        p->setNext(new_node);
      }
    }
  }
  void removeNode(Node& node) {
    for(Node *p0 = &head, *p1 = head.getNext();
        p1 != NULL;
        p0 = p1, p1 = p1->getNext() ) {
      if ( p1 == &node ) {
        p0->setNext(p1->getNext());
      }
    }
  }
  Node* getHead() { return &(this->head); }
  friend std::ostream&
      operator<<(std::ostream& os,
                 LinkedList& l);
private:
  Node head;
};

std::ostream&
    operator<<(std::ostream &os,
    LinkedList& l) {
  for (Node* p = l.getHead()->getNext();
       p != NULL;
       p = p->getNext() ) {
    os << *(p) << " -> ";
  }
  os << "NULL";
}
```

2.2 キュー (Queue) とスタック (Stack)

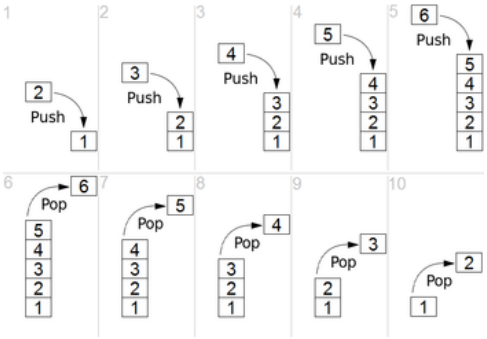
キューは基本的なデータ構造の一つでデータを先入れ先出し (FIFO:First In First Out) するものである．キューにデータを入れることを Enqueue，キューからデータを取り出すことを Dequeue という．キューの変形として先頭と末尾の両端から入出力を行えるものを両端キュー (double-ended queue) と呼び，後入れ先出し (LIFO:Last In First Out) なものをスタック (stack) と呼ぶ．スタックにデータを追加することを Push，スタックからデータを取り出すことを Pop という．

キューは片方向リストや双方向リストで実装できる．双方向リストの場合，直接，先頭と末尾にデータの追加と削除を行うことができるため，それぞれの計算コストは $O(1)$ となる．一方，片方向リストの場合，先頭あるいは末尾のどちらかしか直接的には追加と削除ができない．末尾のデー

タを操作するためには，リストの先頭から末尾まで辿り，そこで操作する必要があり，$O(n)$ の計算コストが必要となる．しかしながら，最後のノードを指し示すポインタを用意することで，$O(1)$ の計算量とすることができる．



https://en.wikipedia.org/wiki/Queue_(abstract_data_type)



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Queue と Stack 実装の例を次に示す．ここでは，前述の LinkedList を用いている．

LinkedList は片方向リストのため，以下の dequeue は片方向リストの先頭のデータを削除しているが，enqueue では，まず一回，片方向リストの最後まで走査して，そこにデータを追加している．

```
class Queue {
 public:
  int dequeue() {
    if ( queue.getHead()->getNext() == NULL ) {
      return -1;
    }
    int ret = queue.getHead()->getNext()->getData();
    queue.removeNode(*(queue.getHead()->getNext()));
    return ret;
  }
  void enqueue(int data) {
    Node* node = new Node(data);
    // find tail node
    Node* tail = queue.getHead();
    while ( tail->getNext() != NULL ) {
      tail = tail->getNext();
    }
    // inset data to the last
    queue.insertNode(*tail, node);
  }
  friend std::ostream& operator<<(std::ostream& os, Queue& s);
 private:
  LinkedList queue;
};
std::ostream &operator<<(std::ostream &os, Queue& s) {
  os << s.queue;
}
```

Stack の実装を以下に 2 通り示す．1 つめは LinkedList を用いたものである．

```
class Stack {
 public:
  int pop() {
    if ( stack.getHead()->getNext() == NULL ) {
      return -1;
    }
    int ret = stack.getHead()->getNext()->getData();
    stack.removeNode(*(stack.getHead()->getNext()));
    return ret;
  }
  void push(int data) {
    Node* node = new Node(data);
    stack.insertNode(*(stack.getHead()), node);
  }
  friend std::ostream& operator<<(std::ostream& os, Stack& s);
 private:
  LinkedList stack;
};
std::ostream &operator<<(std::ostream &os, Stack& s) {
  os << s.stack;
}
```

一方，Stack の場合，例えば最大のスタックの深さをあらかじめ決めていれば，以下のように配列を利用することも可能である．このプログラムは tail で現在のスタックの深さを保持している．ただし実際的な利用では，tail がマイナスになったり，スタックの深さを超えるようなエラー判定をする必要がある．

```
class Stack2 {
 public:
  Stack2() {
    tail = 0;
  }
  int pop() {
    tail--;
    return stack[tail];
  }
  void push(int data) {
    stack[tail] = data;
    tail++;
  }
  friend std::ostream& operator<<(std::ostream& os, Stack2& s);
 private:
  int stack[10];
  int tail;
};
std::ostream &operator<<(std::ostream &os, Stack2& s) {
  for(int i = 0; i < s.tail; i++){
    os << s.stack[i] << " ";
  }
}
```

```
int main() {
  Stack stack;
  stack.push(12); // std::cout << stack << std::endl;
  stack.push(99); // std::cout << stack << std::endl;
  stack.push(37); // std::cout << stack << std::endl;

  std::cout << "stack : ";
  std::cout << stack.pop() << " "; // std::cout << stack << std::endl;
  std::cout << stack.pop() << " "; // std::cout << stack << std::endl;
  std::cout << stack.pop() << std::endl; // std::cout << stack << std::endl;

  Queue queue;
  queue.enqueue(12); // std::cout << queue << std::endl;
  queue.enqueue(99); // std::cout << queue << std::endl;
  queue.enqueue(37); // std::cout << queue << std::endl;

  std::cout << "queue : ";
  std::cout << queue.dequeue() << " "; // std::cout << queue << std::endl;
  std::cout << queue.dequeue() << " "; // std::cout << queue << std::endl;
  std::cout << queue.dequeue() << std::endl; // std::cout << queue << std::endl;

  Stack2 stack2;
  stack2.push(12); // std::cout << stack2 << std::endl;
  stack2.push(99); // std::cout << stack2 << std::endl;
  stack2.push(37); // std::cout << stack2 << std::endl;

  std::cout << "stack2 : ";
  std::cout << stack2.pop() << " "; // std::cout << stack2 << std::endl;
  std::cout << stack2.pop() << " "; // std::cout << stack2 << std::endl;
  std::cout << stack2.pop() << std::endl; // std::cout << stack2 << std::endl;
}
```

2.3 探索アルゴリズム

2.3.1 線形探索 (Linear Search)

データの探索とはあるデータ集合から特定のデータを見つける処理である．一番簡単な方法は先頭から順番に比較していく処理であり，線形探索 (Linear Search) と呼ばれる．計算オーダは $O(N)$ となる．

```
num = [192, 211, 391, 458, 606, 775, 892, 954, 989, 998]
def lsearch(n, data):
    for x in data:
        if x == n:
            return True
    return False

print lsearch(192, num)
print lsearch(193, num)
```

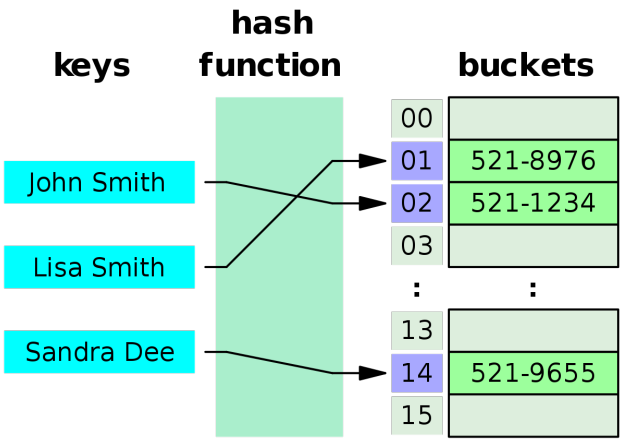
2.3.2 二分探索 (Binary Search)

二分探索 (Binary Search) はデータがソートされていれば，探索する区間を半分に分割しながら調べていくことで，計算量オーダを $O(logN)$ とするアルゴリズムである．ソートのアルゴリズムを適した物を選択すれば線形探索よりも効率が良い．

```
def bsearch(n, data):
    low = 0
    high = len(data) - 1
    while low <= high:
        middle = (low + high) // 2  # Use "//" to return the closest integer value
        if n == data[middle]:
            return True
        elif n > data[middle]:
            low = middle + 1
        else:
            high = middle - 1
    return False

print bsearch(192, num)
print bsearch(193, num)
```

2.3.3 ハッシュ法 (Hashing)

ハッシュ法ではハッシュ表 (hash table) と呼ばれるキー (key) と値 (value)，からなるデータを格納する表と，データを数値に変換するハッシュ関数 (hash function) が必要になる．value の部分のデータは複数でもよく，これをバケットと呼ぶことがある．データ数を N として，hash 関数はデータを 0 から B-1 の数値に変換し，ハッシュ表の対応する key の値 (value) としてデータを格納する．



https://en.wikipedia.org/wiki/Hash_table

異なるデータでも同じじハッシュ値が生成される場合を，ハッシュ値の collision と呼ぶ．チェイン法では同じハッシュ値をキーとする値をリストとして格納する方法であり，オープンアドレス法では空いている key が見付かるまで違うハッシュ関数を適用する方法である．

チェイン法の計算量オーダは，キーを探すのに $O(1)$，元のデータ数 N, ハッシュ関数によって，このデータが B に均等に分割されるとすると，線形リストの探索に $O(1+N/B)$ かかる．

これは，データ数 N に対して，バケット数 B が十分大きい場合は，N/B は定数として考えることが出来るため $O(1)$ となる．一方，データ数 N に対して，バケット数 B が小さい場合は $O(N)$ となる．

一方，オープンアドレス法では 1 バケット，1 データを基本とし，複数のデータが同じハッシュ値を有する場合は衝突 (Collision) と呼ぶが，オープンアドレス法の場合は，別のハッシュ値を見つける必要があるこれを再ハッシュ(rehashing) と呼ぶ．

以下のコードはチェイン法の例になっている．

```
def hsearch(n,data):
    h = n%10
    if h not in hash_table:
        return False
    return lsearch(n, hash_table[h])

hash_table = {}
def make_hash(num):
    for n in num:
        h = n%10
        if h not in hash_table:
            hash_table[h] = []
        hash_table[h].append(n)

make_hash(num)
print hsearch(192, num)
print hsearch(193, num)
```

2.4 ソートアルゴリズム

要素の列（リスト）を与えられた順序に従って並べ替えること．比較方法さえ決めれれば，数字に限らず，文字列等もソートの対象になる．代表的なソートアルゴリズムの例として選択ソート，バブルソート，ヒープソート，マージソート，クイックソートを紹介する．

ソートのプログラムを示す前に，まずはソートするためのデータを生成するプログラムを作ろう．

Python では以下のようにして要素数 100 のランダムな値を持つ配列を作ることが出来る．

```
import random
num = [random.randint(1,100) for i in range(100)]
```

2.4.1 選択ソート

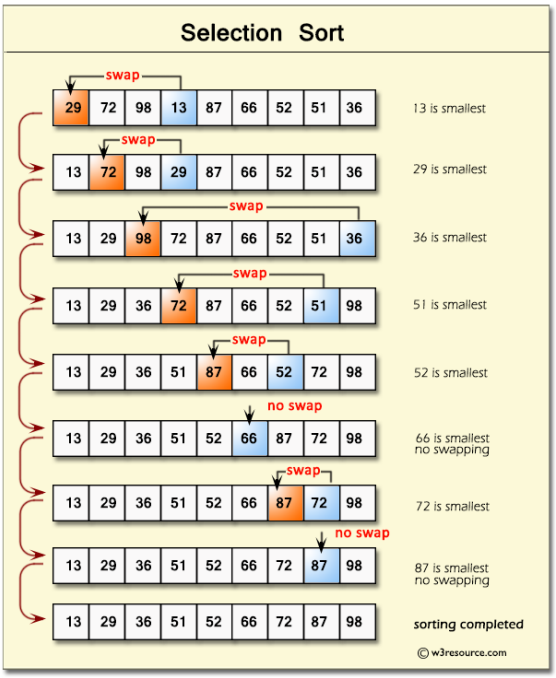
選択ソートは最も単純なアルゴリズムであり，n 個の要素を昇順に並べる場合は以下になる．

1. n 個の中から最も小さい数字を探し，それを 1 番目の数字と入れ替える．
2. 残りの n-1 個の中から最も小さい数字を探し，それを 2 番目の数字と入れ替える．
3. この処理を n-1 回繰り返す．

これにより，要素の配列中で一番小さい値を探し，1 番目の要素と交換する．次に 2 番目以降の要素の配列から一番小さい値を探し 2 番目の要素と交換する．これをデータ列の最後の 1 つ手前まで繰り返している．最後より 1 つ手前までの繰り返しでよいのは，一つ前まで交換済みであれば最後は必ず最大値になるからである．選択ソートプログラムを以下に示す．

Python で同様のプログラムを書いた場合は以下になる．

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import random

def selection_sort(num):
    for i in range(len(num)):
        min = num[i] # 仮の最小値
        min_pos = i  # 仮の最小値の場所
        for n in range(i+1, len(num)):
            if num[n] < min : # 比較対象の数字が仮の最小値より小さければ、仮の最小値を更新
                min = num[n]
                min_pos = n
        tmp = num[i] # 最小値と最初の数を入れ替え
        num[i] = min
        num[min_pos] = tmp
    return num

if __name__ == '__main__' :
    num = [random.randint(1,100) for i in range(10)]
    selection_sort(num)
    print(num)
```



https://www.w3resource.com/php-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-4.php より

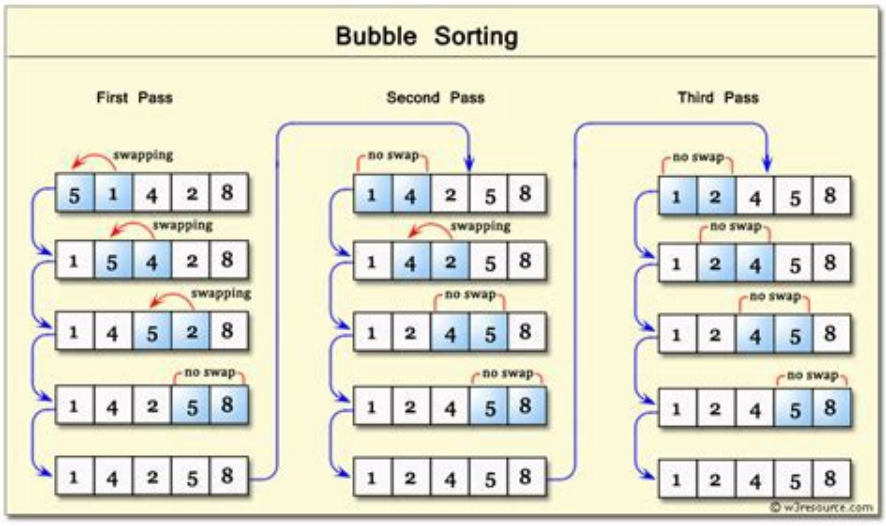
2.4.2 バブルソート

選択ソートでは min, min_pos という変数で仮の最小値とその場所を覚えていたが，これを覚えなくてもよいように改良したのがバブルソートである．そのアルゴリズムは以下になる．

1. n 個の数の中から最小のものを 1 番目にする．その際，以下のようにする．
   - n 番目と n-1 番目を比較し n 番目の方が小さければ入れ替える．
   - n-1 番目と n-2 番目を比較し n-1 番目の方が小さければ入れ替える．

   同様の処理を n-1 回繰り返すと 1 番目に n 個の中から最も小さい数字がくる．
2. 1 番目を除いた n-1 個の数に対して上記の作業を行う．その結果として 2 番目に n-1 個の中から最小の数字（つまり n 個の中で 2 番目の小さい数字）がくる．
3. 同様の処理を n-1 回繰り返す．

```
def bubble_sort(num):
    for i in range(len(num)):
        for n in range(len(num)-1,i,-1):
            if num[n] < num[n-1] : # 比較対象の数字が一つ前の数字より小さければ入れ換える
                tmp = num[n]
                num[n] = num[n-1]
                num[n-1] = tmp
    return num
```



https://python.plainenglish.io/python-sorting-algorithms-bubble-sort-32fda646951c より

2.4.3 ヒープソート

選択ソートとバブルソートは計算量が $O(N^2)$ となるので，データが増えた際の実行時間が非常にかかってしまう．この弱点へ対応したのがヒープソートである．ヒープソートは木構造を使うことで計算量を $O(NlogN)$ に抑えている．

ヒープとは下の図にあるように二分木（一つ節から伸びる枝の本数が 2 本以下の木）を持ったデータ構造であり，(1) 子要素は親要素より常に大きいか等しい (heap property)，(2) 深さ n の要素がすべて使われるまで深さ n+1 の要素は作成しない (shape property)，という制約を持ったものである．これまでの選択ソートとバブルソートはリストを使ったアルゴリズムであるが，ヒープソートはヒープを使ったアルゴリズムとなっている．

ヒープに対してデータの追加と削除をして，かつ，上記の heap property を保つためのための手順を示す．

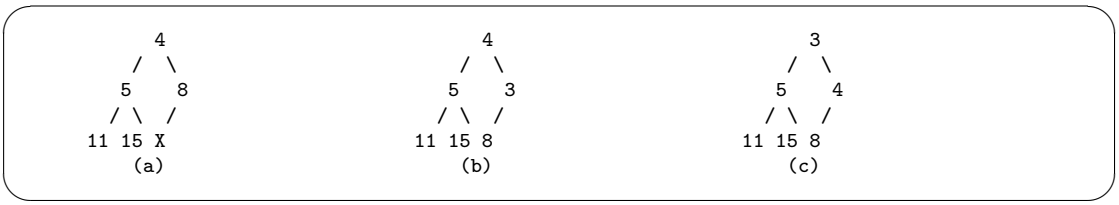
ヒープへのデータの追加

既に存在しているヒープに対してデータを追加する操作は

1. ヒープの最下層に要素を追加する
2. 追加した要素とその親を比較する．正しい順序で並んでいるならば停止する
3. もし順序が正しくないならば親と追加要素を交換して 2 に戻る．

となる．これは高々$O(logN)$ の計算時間である．

簡単な例を示そう．



まず (a) のヒープに対して 3 という要素を付け加えたい場合，X の部分に 3 を追加する．次に，追加した要素とその親 (8) を比較し，正しくないためこの 2 つの要素を交換する．その状態が図 (b) である．さらに，この 3 と，その親の 4 を比べ交換した結果が図 (c) である．これでヒープの追加が完了する．

ヒープからデータの削除

一方ヒープからのデータの削除の操作は以下になる．

1. ヒープのルートを削除し末端の要素をルートとさし替える．
2. 入れ替えた要素とその子を比較する．正しい順序で並んでいるならば停止する
3. もし順序が正しくないならば子（より小さいほうの子）と交換して 2 に戻る．

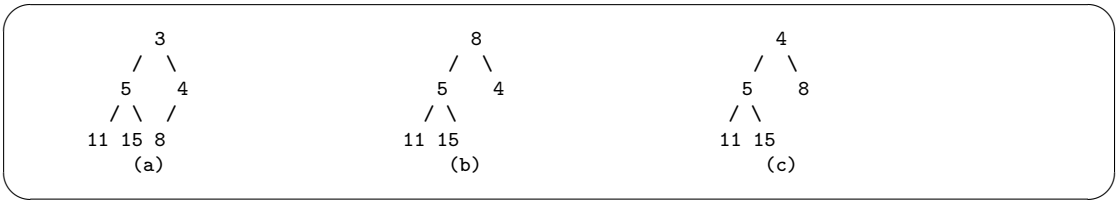
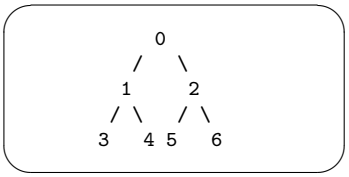


図 (a) が初期状態である．ここからルート（最小要素）である 3 を削除して 8 をルートに持ってきたの図 (b) である．8 とその子である 5,4 ををを比べる 8 のほうが大きく heap property が保たれていない．そこで，より小さいほうの 4 と 8 を交換した結果が図 (c) であり，これが結果である．

配列によるヒープの実装

ヒープはこのように木構造であらわすことが出来るが，計算機上ではデータの配列だけで表現できるという利点がある．つまり，下の図で節の数字が配列の添字であり，a[n] の子は a[2n+1] と a[2n+2] であり，a[n] の親は a[floor((n-1)//2)] となる．



ヒープソートのアルゴリズム

ヒープソートのアルゴリズムは，ヒープから，

1. 配列から要素を取り出しヒープに追加する．すべての要素が追加されるまで繰り返す．
2. ヒープのルート（最小値）を取り出しリストに追加．すべての要素を取り出すまで繰り返す．

という 2 段階となる．以下がそのプログラムであり `heap_sort(num)` として呼び出す．

```
def heap_add(num, c):
    while True:
        p = (c-1)//2 # 親のインデックスを計算 Use "//" to return the closest integer value
        if p < 0 :
            break
        # 親の方が小さい場合は break
        if num[p] <= num[c]:
            break
        #num[p], num[c] = num[c], num[p]
        tmp = num[p]; num[p]=num[c]; num[c]=tmp
        c = p # 親ノードを新たな子ノードにする

def heap_del(num, length, p):
    while True:
        c = p*2+1 # 子のインデックスを計算
        if c >= length:
            break
        if c+1 < length and num[c+1] <= num[c] :
            c = c+1
        # 親の方が小さい場合は break
        if num[p] <= num[c]:
            break
        tmp = num[p]; num[p]=num[c]; num[c]=tmp
        p = c # 子ノードを新たな親ノードにする
```

```
def heap_sort(num):
    for i in range(1,len(num)):
        # 最初のヒープを作る
        heap_add(num, i)
    for i in range(len(num)):
        tmp = num[len(num)-1-i] # num[-1]
        num[len(num)-1-i] = num[0]
        num[0] = tmp
        heap_del(num, len(num)-1-i, 0)
    # 逆順にする
    for i in range(len(num)//2):  # Use "//" to return the closest integer value
        tmp = num[i]
        num[i] = num[len(num)-i-1]
        num[len(num)-i-1] = tmp
```

2.4.4 マージソート

マージソートとはまず列を分割し，この分割された列のそれぞれに対してソートを行い，最後にソートされた列を併合 (merge:マージ) して元の列をソートする方法である．これもまた計算量を $O(NlogN)$ に抑えることが出来る方法である．そのアルゴリズムは以下になる．

1. 列を二分割する
2. それぞれの列をソートする．このときのソートは自分自身を呼び出し再帰的に処理を行う．
3. 2 つのソートされた列をマージする

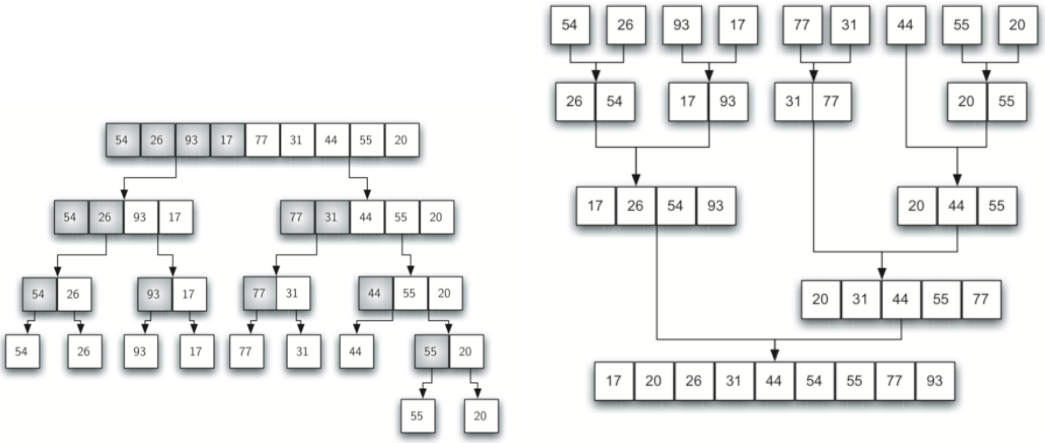
以下にマージソートのプログラムを示す．`merge_sort(num)` は `merge_sort_impl(num, 0, len(num)-1)` として呼び出す．ソートは列を中央値で区切り 2 つの列とする．分割した列の要素数が 1 より大きい場合はもう一度ソートプログラムを呼び出し列の要素が 1 になるまで再帰的に分割する．

マージはソート済みの 2 つの列 (列 `i`, 列 `j`) の先頭を比較し小さいほうを取り出していく処理となる．紹介したプログラムでは `i,j` がそれぞれ 2 つの列の最初の要素を指し示し，`k` が新しいソート済みの列 (列 `k`) の先頭を指し示す．列 `k` は `tmp` という配列に値を保持する．

列 `i` の最後は `right` で，列 `j` の最後は `l` で指し示される．

7 行目から 14 行目はそれぞれの列の先頭の要素を比較し，小さいほうをソート済みの列 `k` の先頭に代入する．小さいほうの列の先頭は一つづらす．15 行目から 18 行目は列 `j` が先に終わった場合（列 `i` に残りがあった場合) にその要素を列 `k` に追加する処理であり，19 行目から 22 行目はその逆である．23 行目から 24 行目は列 `k` の要素を元の列に書き戻している．

```
 1 def merge(num, left, right, size):
 2     tmp = [0]*(left+size)
 3     i = left;
 4     j = right;
 5     k = left;
 6     l = left + size
 7     while i < right and j < l:
 8         if num[i] < num[j]:
 9             tmp[k] = num[i];
10             i+=1
11         else:
12             tmp[k] = num[j];
13             j+=1
14         k+=1
15     if i < right :
16         for h in range(i, right):
17             tmp[k] = num[h];
18             k+=1
19     if j < l :
20         for h in range(j, l):
21             tmp[k] = num[h];
22             k+=1
23     for h in range(left, l):
24         num[h] = tmp[h]
25
26 def merge_sort_impl(num, left, right):
27     if left < right :
28         middle = (right + left)//2  # Use "//" to return the closest integer value
29         merge_sort_impl(num, left, middle)
30         merge_sort_impl(num, middle+1, right)
31         merge(num, left, middle+1, right-left+1)
32
33 def merge_sort(num):
34     merge_sort_impl(num, 0, len(num)-1)
```



https://stackoverflow.com/questions/29587752/merge-sort-the-recursion-part より

2.4.5 クイックソート

クイックソートはマージソートと同様に分割しソートするアルゴリズムである．マージソートは分割後の各列の要素数が均等になるよう中央で分割したが，クイックソートは適当な基準値と各要素と比較し，基準値より小さい要素は前方に，大きい要素は後方に移動させる方法である．

1. 基準値（ピボット）を選択する[2]．
2. ピボットより小さい値を前方に大きい値を後方に移動する
3. 2 分割された各々の列をそれぞれソートする．このソートは自分自身を呼び出す．

基準値の選択では，先頭の値，先頭と末尾の値の平均，配列の真ん中の値，先頭の値，末尾の値，配列の真中の値のメディアン等がある．

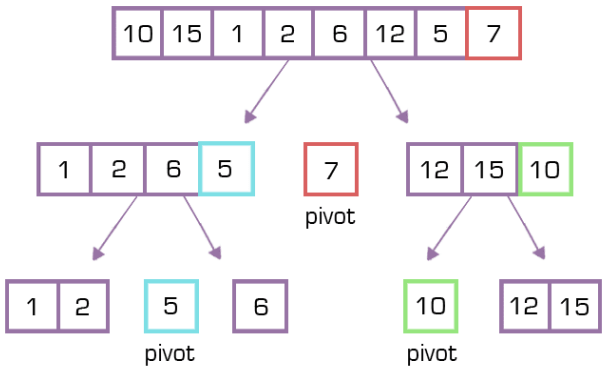
以下は，先頭と末尾の値の平均値で分割したプログラムの例である．2 行目でピボットの値は x に代入されている．6 行目で先頭から順に要素を調べ，x 以上のものがあればその位置を i に覚えておき，8 行目では最後から順に要素を調べ，x 以下のものがあればその位置を j に覚えておく．12 行目でこの 2 つの要素を入れ替えている．これは，`tmp = num[i]; num[i] = num[j]; num[j] = tmp` の書き換えである．

10 行目は i が j 以上になれば，繰り返しをやめている．15-18 行目で要素を 2 分割し，要素数が 1 の領域が出来るまでソートを繰り返す．

平均的な計算量は $O(NlogN)$ であるが，最悪の場合 (分割が要素 1 の列と残りという状況など) は $O(N^2)$ となるが，たいていの場合においてクイックソートは早いといわれている．

```
 1 def quick_sort_impl(num, first, last):
 2     x = (num[first] + num[last])//2  # Use "//" to return the closest integer value
 3     i = first
 4     j = last
 5     while True:
 6         while num[i] < x:
 7             i+=1
 8         while x < num[j]:
 9             j-=1
10         if i >= j:
11             break
12         num[i], num[j] = num[j], num[i]
13         i+=1;
14         j-=1
15     if first < i - 1:
16         quick_sort_impl(num, first, i - 1)
17     if j + 1 < last:
18         quick_sort_impl(num, j + 1, last)
19
20 def quick_sort(num):
21     quick_sort_impl(num, 0, len(num)-1)
```

[2]連立方程式を解く LU 分解でもピボット選択という言葉が出てくる．



https://dev.to/mwong068/quick-sort-in-ruby-2302 より

2.4.6 各ソートアルゴリズムの比較

ここまでのアルゴリズムを比較する表は以下のようになる．

	平均計算時間	最悪計算時間	メモリ使用量
選択ソート	-	$O(N^2)$	$O(1)$
バブルソート	-	$O(N^2)$	$O(1)$
ヒープソート	-	$O(NlogN)$	$O(1)$
マージソート	-	$O(NlogN)$	$O(N)$
クイックソート	$O(NlogN)$	$O(N^2)$	$O(logN)$

また，各種ソートアルゴリズムの比較をするためのプログラムを書くと以下のようになる．各自プログラムをかいて速度を比較してほしい．要素数が 1000 ぐらいまでは一瞬でソートがおわるだろう．要素数が 10000 でもアルゴリズムによっては高速にソートが出来ている．

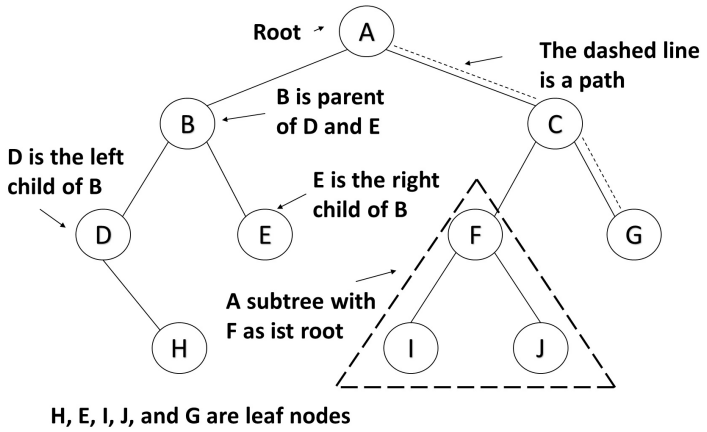
```
import time
def run_sort(f):
    num = [random.randint(1,1000) for i in range(10000)]
    start = time.perf_counter()  # https://docs.python.org/3.7/library/time.html#time.clock
    # call function
    f(num)
    elapsed_time = time.perf_counter() - start
    print("%20s : elapsed time %f [sec]"%(f.__name__, elapsed_time))

run_sort(selection_sort)
run_sort(bubble_sort)
run_sort(heap_sort)
run_sort(merge_sort)
run_sort(quick_sort)
```

3 ツリー (Tree) と走査（Traverse）・探索 (Search) アルゴリズム

本章では，アルゴリズムとデータ構造の一つとして木構造を扱う．

木構造はノード (node / vertex： 頂点) の集合であり，各ノードは 0 個以上の子ノード (Child Node) を持ち，子ノードは木構造内では下方に存在するとする．逆に子ノードは一つの親ノード (Parent Node) を持っている．あるノードから見て，同じ親を持つノードを兄弟ノード (Sibling Node) という．根 (root) ノード親ノードを持たないノードで，一つの木構造の中には根は一つである．根ノードを深さ 0 とすると，その子ノード（達）は深さ 1 と呼ぶ，そのまた子ノードは深さ 2 である．また，子ノードを持たないノードを葉ノード (Leaf Node) と呼ぶ．

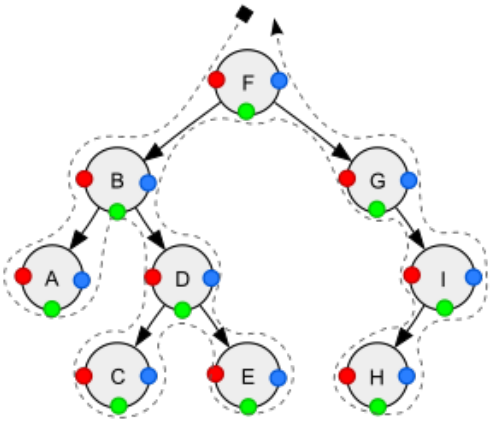
また，各ノードが子ノードを最大２つしか持たない木構造を二分木 (Binary Tree）と呼ぶ．さらに，深さが N の二分木において N-1 のノードが全て埋まっている場合を平衡二分木 (self-balancing binary tree) と呼ぶ．ヒープソートで用いたデータ構造は平衡二分木である．



https://codersite.dev/tree-data-structure-binary-search-tree/ より

3.1 木の走査 (Traverse)

木の走査 (Traverse) とは，全てのノードを一回づつ訪れる処理であり，深さ優先と幅優先が存在する．深さ優先は現在のノードから，深さ方向（子のノードの方向）へと走査を行い，できるだけ深い（親から遠い）方のノードを，兄弟 (sibling 同じ深さにある他のノード）より優先的に走査する．




```
Depth-first traversal (dotted path) of a binary tree:
Pre-order (node access at position red ):
    F, B, A, D, C, E, G, I, H;
In-order (node access at position green ):
    A, B, C, D, E, F, G, H, I;
Post-order (node access at position blue ):
    A, C, E, D, B, H, I, G, F.
```

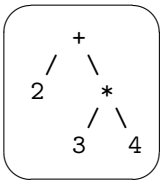
Sorted_binary_tree_ALL.png より

走査法は根から走査をはじめ，ノードを走査する順序によって以下の３つに分類される．

pre-order 根ノードを走査する，もしあれば，左の部分木を走査する，もしあれば右の部分木を走査する．

in-order もしあれば，左の部分木を走査する，根ノードを走査する，もしあれば右の部分木を走査する．

post-order もしあれば，左の部分木を走査する，もしあれば右の部分木を走査する．根ノードを走査する，



走査の応用として，数式の構文木からポーランド記法[3]，逆ポーランド記法，中置記法への変換がある．ここでは， `2 + (3 * 4)` を考えてみる．これの構文木で表すと，右上のようになる．これを，隣接リストとして表し，それぞれの走査アルゴリズムを実装した例が以下のプログラムである．

```
# Using a Python dictionary to act as an adjacency list
graph = {
    '+' : ['2','*'],
    '2' : [],
    '*' : ['3', '4'],
    '3' : [],
    '4' : [],
}

# define pre/in/post order
def pre_order(graph, root):
    res = []
    if root:
        res.append(root)
        for node in graph[root]:
            res = res + pre_order(graph, node)
    return res

def in_order(graph, root):
    res = []
    if root:
        res = res + in_order(graph, graph[root][0] if len(graph[root])>=1 else None)
        res.append(root)
        res = res + in_order(graph, graph[root][1] if len(graph[root])>=2 else None)
    return res

def post_order(graph, root):
    res = []
    if root:
        for node in graph[root]:
            res = res + post_order(graph, node)
        res.append(root)
    return res

print(pre_order(graph, '+'))
print(in_order(graph, '+'))
print(post_order(graph, '+'))
```

[3] https://en.wikipedia.org/wiki/Polish_notation

それぞれ実行結果が，`['+', '2', '*', '3', '4']`, `['2', '+', '3', '*', '4']`, `['2', '3', '4', '*', '+']` となり，前置記法（ポーランド記法），中置記法，後置記法（逆ポーランド記法）を得ることが出来る．

3.2 木の探索 (Search)

木の中から目的のデータを見つけることを探索 (Search) と呼ぶ．木の探索アルゴリズムの概要を以下に示す．

```
function TREE-SEARCH( tree, goal) returns a node, or failure
    fringe ← INSERT(ROOT(tree, )fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE(fringe)
        if GOAL-TEST(goal, node) then return node
        fringe ← INSERTALL(EXPAND(node, tree), fringe)
```

tree が対象となる木構造，*goal* がこの木の中にあるかどうか調べたいデータとする．

ROOT(*tree*) を木の根を返す手続き，EXPAND(*node, tree*) は，この *node* の子ノードを返す手続である．また，GOAL-TEST(*goal, node*) で，*node* が *goal* に到達したか調べている．

fringe はこれから訪れるノードのリストであり，一般的にキュー (queue) で実装されることが多い．REMOVE(*fringe, fringe*) を *fringe* の先頭からデータを取り出す手続きとして，INSERT(*node*) を *fringe* の最後にデータを追加する手続きとすれば，FIFO のキューとなる．

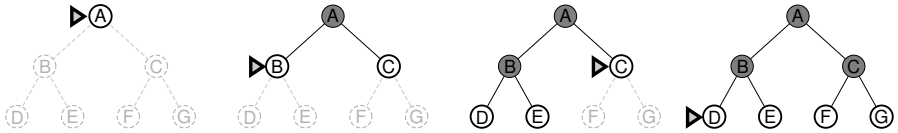
また，REMOVE(*fringe*) を *fringe* の先頭からデータを取り出する手続きとして，INSERT(*node, fringe*) を *fringe* の先頭にデータを追加する手続きとすれば，LIFO のスタックとなる．INSERTALL*node-list, fringe*) はノードのリストを *fringe* に追加する手続きである．fringe からノードを一つ選ぶ処理の違いによって様々な探索アルゴリズムを実現できる．

これから幾つか探索アルゴリズムを紹介する．それぞれの探索アルゴリズムを比較するために以下の４つの基準がある．

- 完全性：解が存在するときに必ずみつかるか
- 最適性：最小経路コストの解を最初にみつけるか
- 時間計算量：解を見つけるための計算時間
- 空間計算量：必要なメモリ量

3.3 幅優先探索 (Breadth-first Search)

幅優先探索は同じレベルのノードを順番に見いくものであり，TREE-SEASRCH アルゴリズムにおいて，FIFO(First-In First-Out) キューを利用することで実現することが出来る．ここでは，展開されたノードを FIFO キューに入れることで，浅いノードがより深いノードより先に取り出され次の訪問先となる．



幅優先探索の様子．白丸はキューに入っているノード，灰色丸は記憶されているノード．
次に展開されるノードにマークが付いている．

Python で記述したプログラム例を以下にしめす．`bfs(graph, 'A', 'D')` として実行出来る．

```
graph = {
    'A' : ['B','C'],
    'B' : ['D','E'],
    'C' : ['F','G'],
    'D' : ['H','I'],
    'E' : ['J','K'],
    'F' : ['L','M'],
    'G' : ['N','O'],
    'H' : [], 'I' : [], 'J' : [], 'K' : [], 'L' : [], 'M' : [], 'N' : [], 'O' : []
}

def bfs(graph, start, target):
    fringe = [start]
    while fringe:
        node = fringe[0]; fringe = fringe[1:] # pop
        print("traverse {}".format(node))
        if node == target:
            print("found target {}".format(node))
            return target
        for n in graph[node]:
            fringe.append(n)
    return None
```

幅優先探索では完全性が満たされる．つまり、解が存在する場合はグラフによらず解をみつけることができる．しかしながらグラフが無限で探索対象の解が存在しない場合は幅優先探索は終了しない。また，最も浅い位置にいるノードをゴールとするため，全てのノード間の遷移のコストが一定であれば，最もコストの少ない経路を見つけることができ，最適性が保たれるといえる．

空間計算量の計算は次のように行う．まず，それぞれのノードが b 個の子を持つとすると，深さ d では b^d 個のノードを持つことになり．深さ d の一番最後のノードがゴールだった場合，この時点では $b^{d+1} - b$ のノードを展開していることになり，必要なメモリ量は

$$b + b^2 + b^3 + ... + b^d + (b^{d+1} - b) = O(b^{d+1})$$

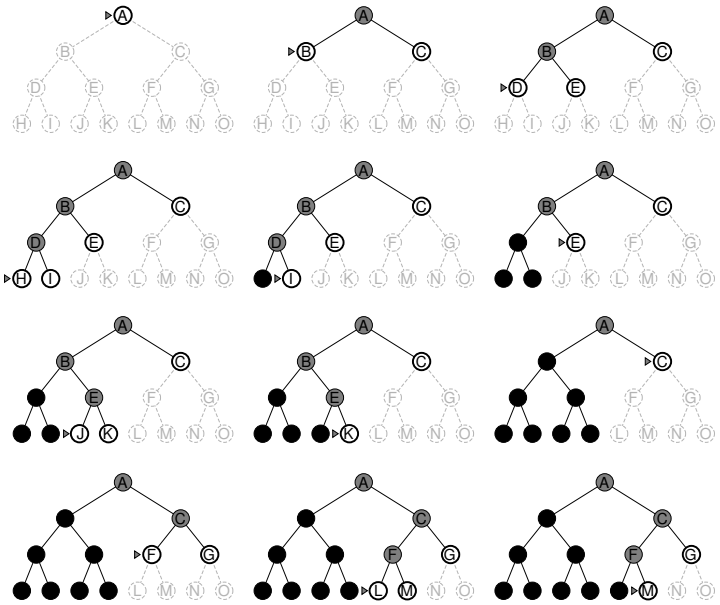
となる．

下の表は b が 10 のときの，それぞれの深さでの展開ノード数と，その時間，メモリを示したものである．時間とメモリは 1 秒に 10000 個のノードを展開できるとし，1 ノードあたり 1000byte 必要と仮定して計算している．

Depth	Nodes	Time	Memory
2	1,100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabyte
6	10^7	19 minutes	10 gigabyte
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabyte
12	10^{13}	35 years	10 petabyte
14	10^{15}	3,523 years	1 exabyte

3.4 深さ優先探索 (Depth-first Searach)

深さ優先探索は最も深いノードを，子がなくなるかゴールに到達するまで展開しつづける戦略である．これは TREE-SEARCH アルゴリズムにおいて LIFO(Last-In First-Out) キューを利用することで実現できる[4]．



深さ優先探索の様子．白丸はキューに入っているノード，灰色丸は記憶されているノード，
黒色は記憶領域に入っていないノード，次に展開されるノードにマークが付いている．

Python で記述したプログラム例を以下にしめす．`dfs(graph, 'A', 'M')` として実行出来る．

```
def dfs(graph, start, target):
    fringe = [start]
    while fringe:
        node = fringe.pop()
        print("traverse {}".format(node))
        if node == target:
            print("found target {}".format(node))
            return node
        for n in reversed(graph[node]): ## added reversed just for text book
            fringe.append(n)
    return None
```

[4]LIFO キューは一般にスタックと呼ばれる．また，深さ優先探索は再帰プログラムとして書くことが出来る．

深さ優先探索は空間計算量にメリットがある．これは，その子孫も含めて展開が済んだノードはメモリから削除することが出来るからである．上の図では黒い丸は記憶する必要がなく，白いノードだけを記憶すればよい．幅優先探索で考察したのと同様にそれぞれのノードが b の子を持ち最大 m の深さをもつツリーを考えた場合，深さ優先探索では最大 $bm+1$ ノードを記憶しておけばよいことになる．前節の表を考えたとき，深さ m=12 の場合で 121KB メモリだけで計算できる．

深さ優先探索の弱点は完全性と最適性が満たされない点である．すなわち，上図で解が C にある場合でも，左側の木を展開し，例えば J もゴールである場合，C ではなく J をゴールとして見つけてしまい最適性が満たされない．また，左側の木が無限の深さを持ち，そこに解が存在しない場合は探索が終了しないため，完全性も保障されない．最悪の場合深さ優先探索は，解が深さ d にあって木の深さが m の場合でも，$O(b^m)$ のノードを展開してしまうため，これ時間計算量になる．これは d ＜＜ m の場合に極端に効率が悪い．

3.5 深さ制限探索 (Depth-limited Search)

深さ優先探索において探索の深さに一定の制限をおき，その弱点を補うのが深さ制限探索である．制限となる深さを l とした場合空間計算量は $O(bl)$ となり時間計算量は $O(b^l)$ となる．

深さの限界は問題の知識によって適切に選択することが出来る．例えば，`'A'` から`'O'` までのアルファベットからなるグラフの問題の場合はノード数は 15 しかないので，もし解があればその長さは最長で 14 となる．また，木の深さは 3 であるため，最適な深さ制限は 3 であるといえる[5]．

Python で記述したプログラム例を以下にしめす．`dlfs(graph, 'A', 'M', 3)` として実行出来る．

```
def dlfs(graph, node, target, depth_limit, limit = 0):
    cutoff_occured = False
    print("traverse {}, level {}".format(node, limit))
    limit = limit + 1
    if node == target:
        print("found target {}".format(node))
        return node
    elif limit > depth_limit:
        return 'cutoff'
    for n in graph[node]: ## added reversed just for text book
        ret = dlfs(graph, n, target, depth_limit, limit)
        if ret == 'cutoff':
            cutoff_occured = True
        elif ret != None:
            return ret
    if cutoff_occured:
        return 'cutoff'
    else:
        return None
```

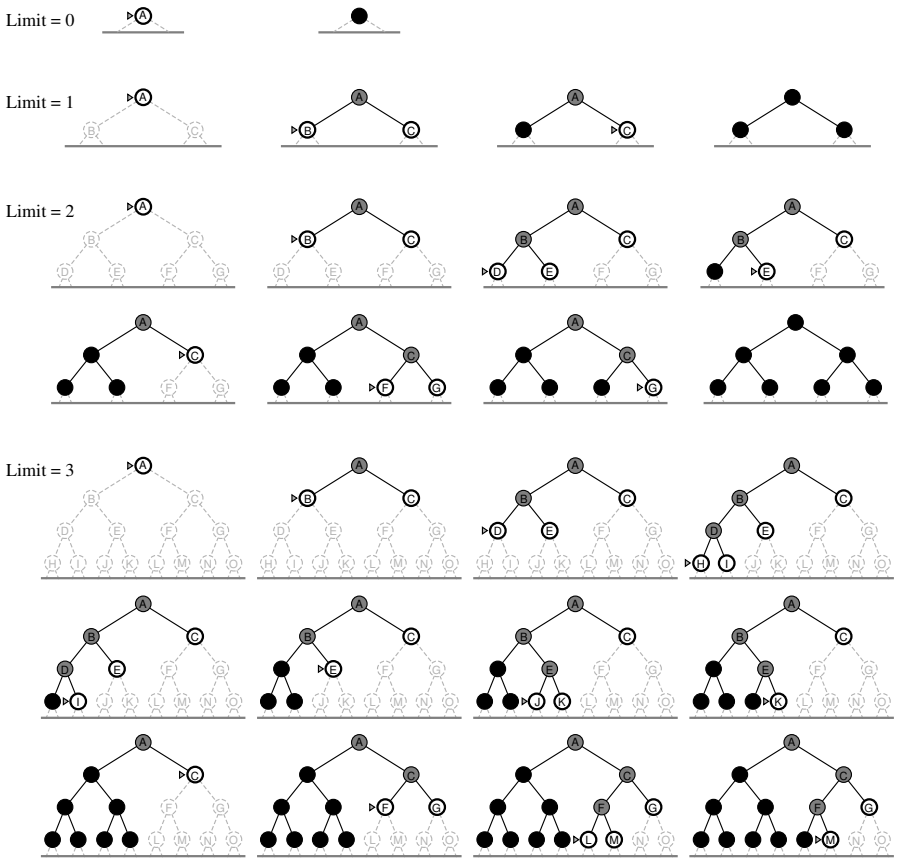
3.6 反復深化探索 (Iterative Deppting Search)

反復深化探索は深さ制限探索を繰り返し呼び出し，その際に最初は深さ制限 1，次は深さ制限 2 と最良の深さ制限を選ぶような振る舞いをする．

[5] しかし，一般に最適な深さは問題を解くまでわからない

プログラムは以下に示すように深さ制限探索を繰り返し呼び出せばよい.

反復深化探索は，幅優先探索と同様に最適かつ完全であり，さらに，空間計算量は $O(bd)$ で深さ優先探索と同様の少ないメモリの消費量となっている.



反復深化探索の様子.

反復深化探索では深さ制限を変更するたびに同じノードを何回も展開するため計算コストが高そうに見える．しかし，実際には深さ d まで探索する場合，深さ d のノードは 1 回の展開がなされるが深さ d-1 のノードは 2 回の展開がなされ，d-n のノードは n 回の展開がなされる，という風になる．これは，ノード数の多い深いノードのほうの展開の数が少なく，ノード数の少ない浅いノードの展開数が少ないため，実際はそれほどコストは高くない.

```
function ITERATIVE-DEEPENING-SEARCH( tree, goal) returns a node
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( tree, goal, depth)
        if result ≠ cutoff then return result
    end
```

深さ d まで探索したときの展開ノード数は

$$N(IDS) = (d)b + (d-1)b^2 + ... + (1)b^d$$

となり，オーダは $O(b^d)$ である．一方，幅優先探索では

$$N(BFS) = b + b^2 + ... + b^d + (b^{d+1} - b)$$

であった．幅優先探索では深さ d を調べる際に深さ d+1 のノードを展開しているが，反復深化探索ではこれをする必要がなく，これが一般に幅優先探索より反復深化探索が高速な理由であり，実際に，b = 10 で，d = 5 の場合，

$$\begin{aligned} N(IDS) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(BFS) &= 10 + 400 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \end{aligned}$$

となる．

ちなみに，深さ優先探索のノード展開数は

$$N(DFS) = 10 + 400 + 1,000 + 10,000 + 100,000 = 111,410$$

となるため，時間計算量は各ノードの子孫の数 b が 10 のときは幅優先探索より 11%多いだけである．また b=2 のときでも約 2 倍になっている．

3.7 各手法の比較

b を各ノードの子の数，d を一番浅い答えの深さ，m を木の深さの最大値，l を深さ制限としたときの各手法の比較を下に示す．

	幅優先	深さ優先	深さ制限	反復深化
完全性	Yes^x	No	No	Yes^x
最適性	Yes^y	No	No	Yes^y
時間	$O(b^{d+1})$	$O(b^m)$	$O(b^l)$	$O(b^d))$
空間	$O(b^{d+1})$	$O(bm)$	$O(bl)$	$O(bd)$

各手法の比較: xb が有限であれば完全，y 各ノード間のコストが一定なら最適

4 おわりに

探索アルゴリズムは完全性，最適性，時間計算量，空間計算量の基準で判断できいる．代表的なアルゴリズムとして以下のものを紹介した．幅優先探索は，探索木が浅いノード順に展開し，完全かつ最適であるが，時間空間計算量が膨大であった．深さ優先探索は探査木の最も深いノードを最初に展開する．これは完全でも最適でもなく，時間計算量も大きいが，空間計算量は小さい．ちなみに，幅優先探索を横型探索，深さ優先探索を縦型探索と呼ぶ．深さ制限探索は深さ優先探索の深さ制限をつけたものであり，これを深さ制限を徐々に大きくとりながら繰り返し計算する反復深化探索は完全かつ最適で，かつ空間計算量も少なく，幅優先探索と深さ優先探索の長所を併せ持ったアルゴリズムである．

宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にでてきたキーワードについて知らなかったもの, 興味のあるものを調べ「学籍番号.txt」としててアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．
画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい．
ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード：ハッシュ関数，破壊的ソート, 構文木，隣接リスト

1. 選択ソート，バブルソートのプログラムを再帰関数として実装せよ．

2. 逆ポーランド記法とは数式において，演算子を最後に置く記法である．例えば以下のように記述する．

```
1 + 2          -> 1 2 +
2 + ( 3 * 4 )  -> 2 3 4 * +
```

逆ポーランド記法で記された式はスタックを用いて以下のように計算出来る．このプログラムを作成せよ．

- 式を表すデータから先頭に値を読み込む.
- 数を読み込んだらスタックに積む (Push する)
- 演算子を読み込んだらスタックから 2 つの数を取出し (Pop し) 演算子を適用する．取り出した順に x, y とすると，y 演算子 x とする．さらにこの計算の結果をスタックに積む (Push する).
- 読み込むデータがなくなったら，スタックから数を取出し (Pop し) 式の結果を得る．

ヒント 1：スタックは以下のようにリストの append/pop で実現できる．

```
>>> stack = []
>>> stack.append(1)
>>> stack.append(2)
>>> stack.append(3)
>>> stack.pop()
3
>>> stack.pop()
2
>>> stack.pop()
1
```

ヒント 2：式は以下のように数値と文字を用いて実現できる．式を順番に読み出し，数値か演算子は以下のようにプログラムできる．

```
data = [2, 3, 4, '*', '+']
for d in data:
  if type(d) == int:
     print("this is int")
  else:
     if d == '+':
        print("this is +")
     else:
        print("other operator")
```

発展的な話題 1：演算子による場合分けをしたくない場合は，以下のように関数を式に入れる方法がある．

```
def mul(x, y): return x * y
def add(x, y): return x + y
data = [2, 3, 4, mul, add]
for d in data:
    if type(d) == int:
        print("input is {}".format(d))
    else:
        print("input is {}(1, 2) = {}".format(d, d(1, 2)))
```

発展的な話題 2：さらに，以下のように無名関数を式に入れると，あらかじめ演算子の種類を決めておく必要も無くなる．

```
data = [2, 3, 4, lambda x, y: x * y, lambda x, y: x + y]
```

3. 反復深化探索のプログラムを実装せよ

‹ 2 / 2 ›

以下の深さ優先探索プログラムがあるときに、反復深化プログラムを書いてください。

```
def dfs(graph, start, target):
  fringe = [start]
  while fringe:
    node = fringe.pop()
    print("traverse {}".format(node))
    if node == target:
      print("found target {}".format(node))
      return node
    for n in reversed(graph[node]): ## added reversed just for text book
      fringe.append(n)
  return None
```

以下に、反復深化深さ優先探索（Iterative Deepening Depth First Search、IDDFS）のPythonコードを示します。

python Copy code

```python
def dls(graph, start, target, depth):
    if depth == 0 and start == target:
        return True
    if depth <= 0:
        return False
    for node in reversed(graph[start]):
        if dls(graph, node, target, depth - 1):
            print("Found target {} at depth {}".forma
            return True
    return False

def iddfs(graph, start, target, max_depth):
    for depth in range(max_depth):
        if dls(graph, start, target, depth):
            return True
    return False
```

このプログラムでは、まず深さ制限付き深さ優先探索（Depth Limited Search、DLS）を実装し、その後でIDDFSを実装しています。IDDFSはDLSを反復的に適用し、各反復で深さ制限を1ずつ増やしています。この方法で、グラフの深さ方向を均等に探索できます。

注意点としては、`max_depth`は事前に決める必要があります。探索対象のグラフが無限に深い場合や、目的地までの深さが未知の場合は、この方法が適用できないかもしれません。また、深さ制限を増やすごとに探索範囲が広がるため、計算時間が大幅に増える可能性があります。

A ソートログラム (C 言語)

ソートのプログラムを示す前に，まずはソートするためのデータを生成するプログラムを作ろう．

以下のプログラム (sort-gen.c) をコンパイルし ./sort-gen -o rand.txt -n 100 のようにして実行すると，100 個の要素からなるデータを rand.txt というファイルに書き出す．

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

void
output_rand(int length, char *fileName) {
  FILE *fo; //ファイルポインタ
  int i;
  //ライトモードでファイルを開く
  if ((fo=fopen(fileName, "w"))==NULL) {
    printf("Can't Open Input File.\n");
    exit(1);
  }
  srand(time(NULL));
  for (i = 0; i < length; i++) {
    //ファイルに書き込む
    fprintf(fo, "%d\n", rand() % 1000);
  }
  fclose(fo);
}
```

```
int
main(int argc, char *argv[]) {
  int c, length = 100;
  char *fname = "rand.txt";

  while ((c = getopt(argc, argv, "n:o:"))
         != -1 ) {
    switch (c) {
    case 'n':
      length = atoi(optarg);
      break;
    case 'o':
      fname = optarg;
    }
  }

  fprintf(stderr, "writing data with %d
        length to %s\n", length, fname);
  output_rand(length, fname);
  return (0);
}
```

ソート用データ書き出しプログラム　sort-gen.c

A.1 選択ソートログラム

選択ソートプログラムは以下のようになる．

```
#include<stdio.h>
#include<stdlib.h>

int *read_array(char *fname, int *num,
                int *len);
void print_array(int num[], int length);
void selection_sort(int *num, int length);

int main(int argc, char *argv[])
{
  int length, *num;
  char *fname = "rand.txt";

  printf("reading from %s\n", fname);
  num = read_array(fname, num, &length);

  print_array(num, length);
  selection_sort(num, length);
  print_array(num, length);

  free(num);
  return (0);
}
```

```
void selection_sort(int *num, int length)
{
  int i, n, tmp;
  int min, min_pos;
  for (i = 0; i < length - 1; i++) {
    print_array(num, length);

    min = num[i]; //仮の最小値の値
    min_pos = i; //仮の最小値の場所
    for (n = i + 1; n < length; n++) {
      if (num[n] < min) { //比較対象の
数字が仮の最小値より小さければ、仮の最小値
をそれにする
        min = num[n];
        min_pos = n;
      }
    }
    tmp = num[i]; //最小値と最初の数を入
れ替え
    num[i] = min;
    num[min_pos] = tmp;

  }
}
```

選択ソートプログラム

ちなみに，このプログラムをよく見ると実は再帰呼び出しで書けることがわかる．是非挑戦してみて欲しい．

また，これらのプログラムをコンパイルするためには以下の関数を集めたファイルと Makefile は以下のようになる．

```
CC = gcc
CFLAGS = -Wall -O4
all: sort-gen sort1
%.o: %.c
        $(CC) $(CFLAGS) -o $@ -c $<
sort1 : sort1.o sortlib.o
        $(CC) -o $@ $^
sort-gen : sort-gen.o
        $(CC) -o $@ $^
clean:
        rm -f sort1 sort-gen sort1.o sort-gen.o sortlib.o
```

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int *read_array(char *fname, int *num, int *len)
{
  FILE *fo;
  int i;
  char str[256];
  if ((fo = fopen(fname, "r")) == NULL) { //リードモードでファイルを開く
    printf("Can't Open Input File.\n");
    exit(1);
  }

  *len = 0; while (fgets(str, 256, fo) != NULL) { (*len) ++; }
  num = (int *)malloc(sizeof(int)*(*len));
  fseek(fo, 0, SEEK_SET);

  i = 0;
  while (fgets(str, 256, fo) != NULL) { //ファイルからデータを読み込む
    num[i] = atoi(str);
    i++;
  }
  return num;
}

void print_array(int num[],int length){
  int i;
  for(i=0;i<length;i++){
    fprintf(stdout, "%d, ",num[i]);
  }
  fprintf(stdout, "\n");
}
```

ユーティリティ関数ファイル sortlib.c

選択ソートを実行した結果を以下に示す．

```
% sort-gen -n 10
writing data with 10 length to rand.txt
% ./sort1
reading from rand.txt
79, 12, 29, 96, 81, 8, 43, 49, 8, 84,
79, 12, 29, 96, 81, 8, 43, 49, 8, 84,
8, 12, 29, 96, 81, 79, 43, 49, 8, 84,
8, 8, 29, 96, 81, 79, 43, 49, 12, 84,
8, 8, 12, 96, 81, 79, 43, 49, 29, 84,
8, 8, 12, 29, 81, 79, 43, 49, 96, 84,
8, 8, 12, 29, 43, 79, 81, 49, 96, 84,
8, 8, 12, 29, 43, 49, 81, 79, 96, 84,
8, 8, 12, 29, 43, 49, 79, 81, 96, 84,
8, 8, 12, 29, 43, 49, 79, 81, 96, 84,
8, 8, 12, 29, 43, 49, 79, 81, 84, 96,
```

A.2 バブルソート

```
void bubble_sort(int *num, int length)
{
  int i, n, tmp;
  for (i = 0; i < length; i++) {
    for (n = length - 1; n > i; n--) {
      if (num[n] < num[n - 1]) {//比較対象の数字が一つ前の数字より小さければ入れ換える.
        tmp = num[n];
        num[n] = num[n - 1];
        num[n - 1] = tmp;
      }
    }
  }
}
```

バブルソートもまた選択ソートと同様に再帰呼び出しで書くことが出来る.

A.3 ヒープソート

```
void heap_add(int *num, int length, int c)
{
  int p, tmp;
  while (1) {
    p = (c-1)/2; // 親のインデックスを計算
    if (c < 0)
      break;
    // 親の方が小さい場合は break
    if (num[p] <= num[c])
      break;
    tmp=num[p];num[p]=num[c];num[c]=tmp;
    c = p; //親ノードを新たな子ノードにする
  }
}

void heap_del(int *num, int length, int p)
{
  int c, tmp;
  while (1) {
    c = p*2+1; // 子のインデックスを計算
    if (c >= length)
      break;
    if (c+1 < length && num[c+1] <= num[c] )
      c = c+1;
    // 親の方が小さい場合は break
    if (num[p] <= num[c])
      break;
    tmp=num[p];num[p]=num[c];num[c]=tmp;
    p = c; //子ノードを新たな親ノードにする
  }
}
```

```
void heap_sort(int *num, int length)
{
  int tmp, i;
  //最初のヒープを作る
   for (i = 1; i < length; i++ ){
    heap_add(num, length, i);
  }
 //ヒープから最小値を抜き配列の後ろからつめて
いく
  for (i = 0; i < length; i++) {
    tmp = num[length-1-i];
    num[length-1-i] = num[0];
    num[0] = tmp;
    heap_del(num, length-1-i, 0);
  }
  //逆順にする
  for (i = 0; i < length/2; i++){
    tmp = num[i];
    num[i] = num[length-i-1];
    num[length-i-1] = tmp;
  }
}

int main(int argc, char *argv[])

  ...(省略)...
  heap_sort(num, length);
  ...(省略)...

}
```

A.4 マージソート

```
 1: void merge(int *num, int left, int right,
 2:            int size)
 3: {
 4:  int *tmp, h, i, j, k, l;
 5:  tmp = malloc((left + size) * sizeof(int));
 6:  i = left;
 7:  j = right;
 8:  k = left;
 9:  l = left + size;
10:  while ((i < right) && (j < l)) {
11:    if (num[i] < num[j]) {
12:      tmp[k] = num[i];
13:      i++;
14:    } else {
15:      tmp[k] = num[j];
16:      j++;
17:    }
18:    k++;
19:  }
21:  if (i < right) {
22:    for (h = i; h < right; h++) {
23:      tmp[k] = num[h];
24:      k++;
25:    }
26:  }
```

```
void merge_sort(int *num, int left,
                int right)
{
  int middle;
  if (left < right) {
    middle = (right + left) / 2;
    merge_sort(num, left, middle);
    merge_sort(num, middle + 1, right);
    merge(num, left, middle + 1,
          right - left + 1);
  }
}
```

```
27:  if (j < l) {
28:    for (h = j; h < l; h++) {
29:      tmp[k] = num[h];
30:      k++;
31:    }
32:  }
33:  for (h = left; h < l; h++) {
34:    num[h] = tmp[h];
35:  }
36:  free(tmp);
37:}
```

A.5 クイックソート

```
 1: void quick_sort
       (int *num, int first, int last) {
 2:  int i, j, x, t;
 3:
 4:  x = (num[first] + num[last]) // 2;
 5:  i = first;
 6:  j = last;
 7:  while (1) {
 8:    while (num[i] < x)
 9:      i++;
10:    while (x < num[j])
11:     j--;
```

```
12:    if (i >= j)
13:      break;
14:    t = num[i];
15:    num[i] = num[j];
16:    num[j] = t;
17:    i++;
18:    j--;
19:  }
20:  if (first < i - 1)
21:    quick_sort (num, first, i - 1);
22:  if (j + 1 < last)
23:    quick_sort (num, j + 1, last);
24}
```

```
# Use "//" to return the closest integer value
```

A.6 各ソートアルゴリズムの比較

```
int main(int argc, char *argv[])
{
  int c, length, *num;
  char *fname = "rand.txt";

  enum {SELECTION, SELECTION2, BUBBLE,
        HEAP, MERGE, QUICK}
     sort_algorithm = SELECTION;
  char *sort_algorithm_str[] = {"SELECTION",
    "SELECTION2", "BUBBLE", "HEAP", "MERGE",
    "QUICK"};

  while ((c = getopt(argc, argv,
          "f:s2bhmq")) != -1 ) {
    switch (c) {
    case 'f':
      fname = optarg; break;
    case 's':
      sort_algorithm = SELECTION; break;
    case '2':
      sort_algorithm = SELECTION2; break;
    case 'b':
      sort_algorithm = BUBBLE; break;
    case 'h':
      sort_algorithm = HEAP; break;
    case 'm':
      sort_algorithm = MERGE; break;
    case 'q':
      sort_algorithm = QUICK; break;
    }
  }
```

```
  fprintf(stderr, "reading from %s\n", fname);
  num = read_array(fname, num, &length);
  print_array(num, length);

  fprintf(stderr, "using %s algorithm\n",
          sort_algorithm_str[sort_algorithm])

  switch (sort_algorithm) {
  case SELECTION:
    selection_sort(num, length);
    break;
  case SELECTION2:
    selection_sort2(num, 0, length-1);
    break;
  case BUBBLE:
    bubble_sort(num, length);
    break;
  case HEAP:
    heap_sort(num, length);
    break;
  case MERGE:
    merge_sort(num, 0, length-1);
    break;
  case QUICK:
    quick_sort(num, 0, length-1);
    break;
  }
  print_array(num, length);
  free(num);
  return (0);
}
```

プログラムの実行時間を測る一番簡単な方法は `time` コマンドを用いることだろう．コマンドを実行すると以下のように表示される．

```
$ ./sort-gen -n 10000000
$ time  ./sort1  -q > /dev/null
reading from rand.txt

real 0m6.270s
user 0m6.100s
sys  0m0.092s
```

```
$ time  ./sort1  -q

（中略）

real8m37.488s
user0m7.800s
sys0m1.540s
```

real はプログラムの呼び出しから終了までにかかった実時間，user はプログラム自体の処理時間（ユーザ CPU 時間），sys はプログラムを処理するために OS が処理をした時間（システム時間）である．`> /dev/null` はプログラムの出力を`/dev/null` と呼ばれるスペシャルファイルに書き出す．これは `write()` しても全てのデータを捨てるものである．これをしない場合はデータの表示自体に時間がかかる．

また計算機の時間計算はその分解能以下の計測は出来ない点も注意しよう．

プログラムの実行時間ではなく，プログラム内である時点からある時点までの時刻を計測したい場合は C 言語では

```
#include <sys/time.h>
{
  struct timeval s, e;
  gettimeofday(&s, NULL);
  /* call function */
  gettimeofday(&e, NULL);
  printf("time = %lf\n", (e.tv_sec - s.tv_sec) + (e.tv_usec - %s.tv_usec)*1.0E-6);
}
```

データを表示するには gnuplot というツールが便利である．例えば，

```
100          0.001
200          0.002
300          0.004
400          0.006
500          0.011
600          0.014
700          0.020
800          0.026
900          0.033
```

というデータが書かれた plot.pl というファイルを用意し

```
$ gnuplot
```

としてプログラムを立ち上げる, コマンドプロンプトが立ち上がったら

```
> plot 'plot.pl' with linespoints title 'selection sort [sec]'
```

とするとグラフが表示されるはずである[6].

コマンドプロンプトから以下の様に実行するとグラフを表示し，ファイルに表示する．

```
$ gnuplot -e "plot 'plot.pl' with linespoints title 'selection sort [sec]'; \
            set terminal png; set output 'plot.png'; replot; pause -1"
```

[6]gnuplot を立ち上げた時に，Terminal type set to 'unknown' と表示され，plot コマンドでもグラフが表示されない場合 sudo apt-get install gnuplot-x11 を実行してみよう