かんたん
最短経路探索講座
feat. Python

BinNスタートアップゼミ#n
担当：出原昇馬（M1）
2019/4/18

もくじ

第一部：Pythonについて

Pythonとは？なぜRだけではだめなのか？という疑問から始まり，具体的な基礎文法などについてちょこっと説明します.

第二部：Pythonについて

羽藤研では道路や鉄道などのネットワークを扱う機会が多くなります．最短経路を探索するいくつかのアルゴリズムと，そのために必要なデータ処理構造をご紹介します.

課題説明：Pythonの最短経路探索

第一部 Pythonについて

インストールから基礎の基礎まで

Pythonとは

- スクリプト言語の名前
- パッケージとよばれる，便利な定義のセットがたくさんあります
- IDE（開発環境：編集しやすくなるソフト）もいろいろ
  Atomとか，**PyCharm**とか

- インストールすべきものは三つ
  ①Python本体，②パッケージ，③IDE
  →ですが，「**Anaconda**」を使えば，
  ①Python本体と②データサイエンスおすすめパッケージ
  を一気にインストールできます！

（補）Anacondaの役割は？

- 大きなデータの分析に使うには，早いことが重要です
- しかし**Pythonはわりと遅めの言語**です ※Rよりは早いです
- そこで，C/C++ といった言語で開発された「拡張モジュール」とつなげて使うと早くなります
- そのようなパッケージをインストールするには，自分のOSに対応するようにコンパイルしてからインストールする必要がありますが，
  Anaconda はコンパイル済みのパッケージを出してくれています

要するに
プログラミングにあまり詳しくなくても
すぐ研究に使えて便利！ということです

RとPython

特徴の比較

- Rはデータ分析関係のコマンドが最初から入っている
  ↔Pythonはパッケージに依存している
- Rは統計寄り
  ↔統計以外ならPythonの方が単純

Rだけだと，今後の研究テーマによっては以下のような問題がでることがあります.

- Rは大規模なデータを扱うには適していない
- 基本のRは並列プログラミングに対応していない
- Rは非常に簡単だがブラックボックス化しやすい

のでPythonもやっておくのがおすすめです！

（補）インストールしてみよう

Python本体と主要パッケージ

- Pythonには歴史があり，大きく分けて2.x系と3.x系のふたつのバージョンがあります
- 2.x系は「古い」ので3.x系を入れましょう
- Anacondaによるインストールの仕方は↓

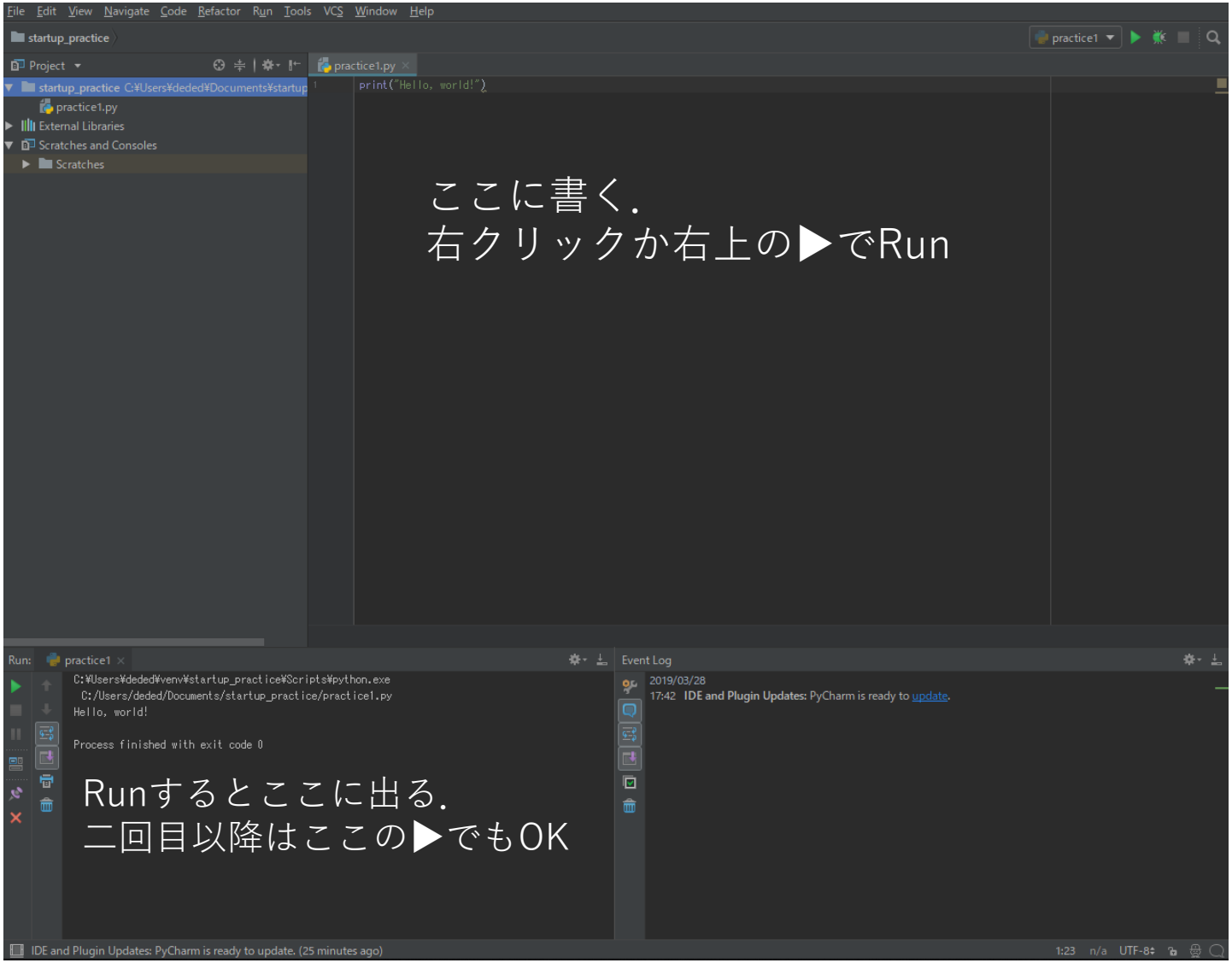
  https://weblabo.oscasierra.net/python-anaconda-install-windows/

IDE

PyCharmにしたい方はこちらを参考に．※以下ではPyCharmで解説します
Community Edition（無料）で事足りると思います．

https://pythondatascience.plavox.info/python%E3%81%AE%E9%96%8B%E7%99 %BA%E7%92%B0%E5%A2%83/pycharm%E3%81%AE%E3%82%A4%E3%83%B3% E3%82%B9%E3%83%88%E3%83%BC%E3%83%AB

PyCharm画面はこんな感じ



注意

エンコーディング

- 3.x系のデフォルトエンコーディングはUTF-8

インデント

- Pythonでは，インデントに意味があります．
- インデントには4文字分のスペースを使うのが一般的です．
  （PyCharmではだいたい自動的に処理してくれます．）

コメント

- 一行におさまるものは，#を先頭につけます．
- 複数行にわたるものは，”””か’’’でコメントを囲みます．

書き方

- カンマ後ろには半角スペースをいれる必要があります．

①プロジェクトをつくります

- 上のタブから
[file]->[New Project]でプロジェクト名を入力
[完了]すると左側にプロジェクトフォルダができます

②ファイルをつくります

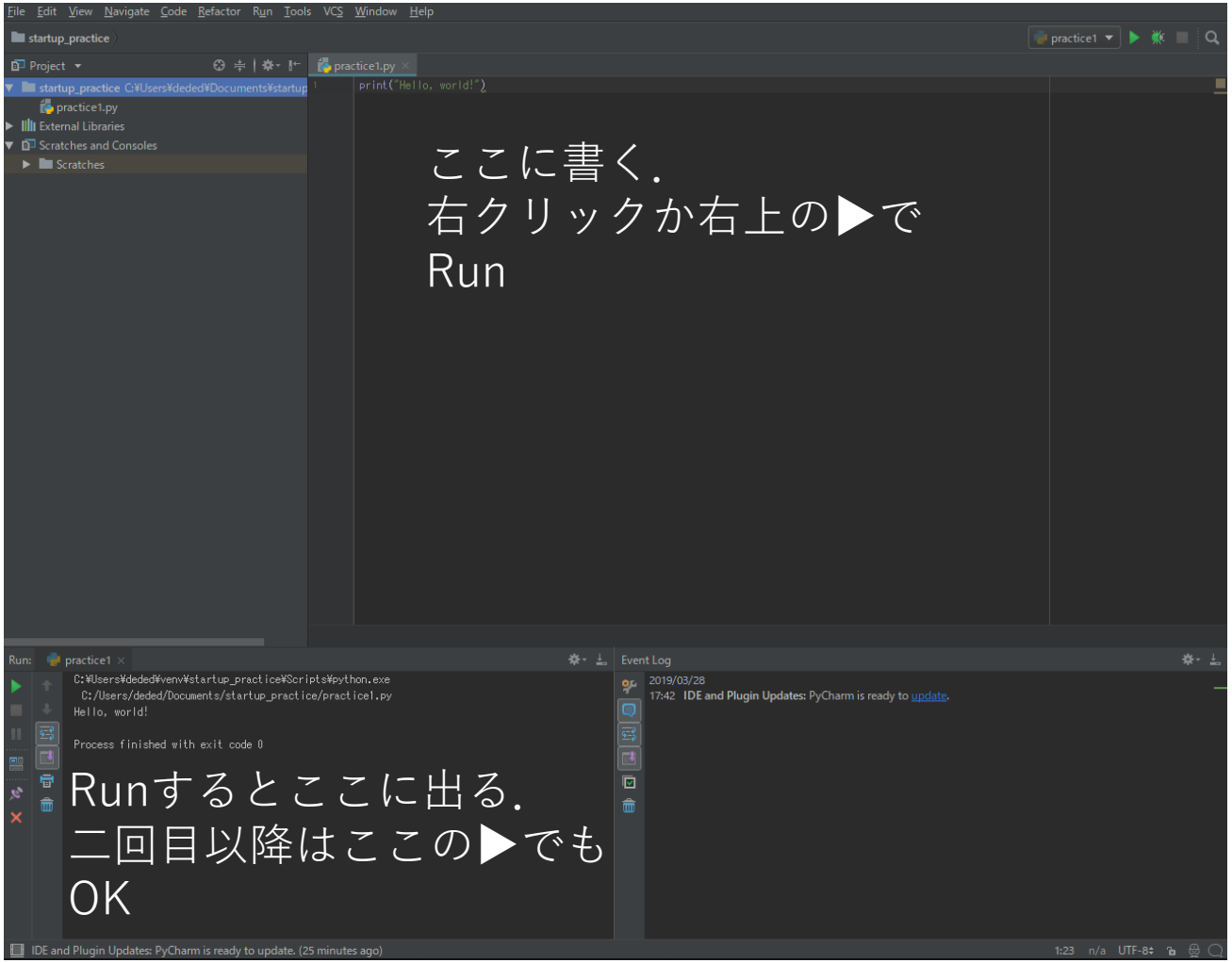
- 左のプロジェクト名を右クリック
- [New]->[Python File]でファイル名を入力，[完了]

③実行します

- コードをかいて，右クリックで出てくるメニューの右上のボタンから
[▶Run]すれば実行されます.
（コンソール左の▶では，直前に実行したファイルを再実行できます）

動かしてみましょう

まずは, print(**“Hello, world!”**)



まず知りたい①識別子

識別子：クラス，関数，変数などの名前

名前の付け方のルール

- アルファベット，アンダースコア_，半角数字が使えます（3.x系では日本語もOK）
- 大文字と小文字は区別されます(例：thetaとTheta)
- 数字は識別子の先頭には使えません
- 予約語（python内で既定の言葉）は使えません
- Ex. False, True, import, if, …
- アンダースコア_を先頭におくと特別な意味がつきます
  (※import()やクラスを利用するとき)

まず知りたい②データ型

Pythonは変数の型を自動で判定します.

型が異なもの同士の演算はできないこともあります.

整数 int
長整数 long
小数 float
複素数 complex ※虚数をjで表すので注意
真理値 bool：True, False

数値

文字列 str：”abcde” “10”
リスト list：[“today”, 200,“next", 250]
タプル tuple：(“today”, 200 , “next”, 250)

数字も“”または‘’の間に書くと
文字列として認識されます

リストは変更可能，タプルは
変更不可だが早いという
違いがあります

シーケンス

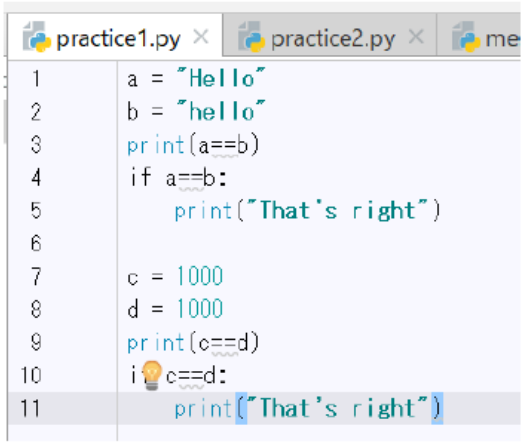
辞書 dict：{"dog":4,"cat": 4,"bird": 2}

値とキーで紐づけできます
ex. “dog”キーで4が呼び出せる

まず知りたい③比較演算子

- 比較演算子　http://www.isl.ne.jp/pcsp/python/python09.html　より

演算子	説明	例	
		プログラム	実行結果
==	左辺と右辺が等しい	1 == 1	True
!=	左辺と右辺が等しくない	1 != 1	False
<	左辺が右辺より小さい	1 < 2	True
>	左辺が右辺より大きい	1 > 2	False
<=	左辺が右辺以下	2 <= 2	True
>=	左辺が右辺以下	2 >= 2	True
is	厳密一致	1 is 1	False
is not	厳密不一致	1 is not 1	True



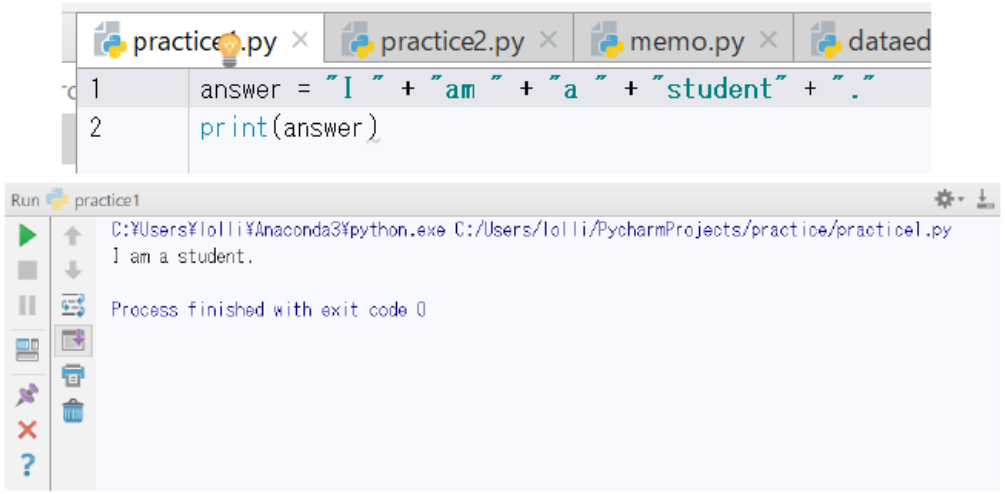
TrueまたはFalseを返します．if文やwhile文に多用します．

まず知りたい④算術演算

- 数値型の算術演算　http://www.isl.ne.jp/pcsp/python/python09.html　より

演算子	説明	例	
		プログラム	実行結果
+	足し算	1 + 2	3
-	引き算	3 - 1	2
*	掛け算	2 * 3	6
/	割り算	4 / 2	2
%	剰余	10 % 3	1
**	べき乗	2 ** 3	8

（補）文字列同士の場合も演算（=連結）ができます.



（補）自学自習

- range()とprint()
- for文, if文, while文, try文
- データ型の判定
- 関数の定義の仕方

まず一冊本を読んで,
なんとなく「こういうことができるらしい」
ということを把握しましょう！

分からなくなったらググるか先輩に聞きましょう！！

たくさん遊んでみるのが大事！！

次に知りたい①モジュールの使い方

- **モジュール**：関数やクラスなどを書いたファイル.

クラスとは，現実世界のものをプログラミング世界に落とし込むときに行う，状態と性質の定義のことです．これをあらかじめ書いておいて，呼び出して使うことで，Pythonでも高速計算が可能です！

- **パッケージ**：モジュールをまとめたもの.
- **ライブラリ**：パッケージをまとめたもの．プログラム群のことをいいます.

モジュールの使い方

```
import モジュール名 as 別名
```

自分で決められて，そのファイル内では
その名前で使うことができます.
pandasをpd，numpyをnpとつけるのが定石です

データ解析系ライブラリ　　配列処理系ライブラリ

(補) 未インストールエラーの時は？

1　まず綴り間違いじゃないか確認する

- File > Settings > project: ○○ > Project Interpreter
  からPackageを確認．ここにあったらたぶん綴り間違い．
- ここになかったら，右上のLatestセルの右横にある＋を押してみる．

2　Available Packagesの確認

- 開いたウィンドウで検索してみる．ここにあったら話は早くて，選択して左下のInstall Package.
- このAvailable Packageにもなかったら，コマンドプロンプトからインストールする必要がある．

3　コマンドプロンプトからのインストール

- cortanaの検索フォームみたいな，画面左下の「ここに入力して検索」というところに「コマンドプロンプト」と入力．出てきたものをクリックしてコマンドプロンプトを立ち上げる．PATHを通していればここからいじれる．
- pip install パッケージ名
  を入れて，エンターキーを押すだけ．こうしたあとで1の手順のようにPycharmから確認するときちんと入っていて，そのパッケージが使えるはず．おしまい！

（※windows10，PyCharm，pipが入っている，
かつPythonをダウンロードしたときにPATHを通している前提）

次に知りたい②csvファイルの読み書き

今回はnumpyを使うものを紹介します！

※pandasにもあります．こちらを使うことも多いです[**自主学習**]

まずは

```
import numpy as np
```

※ファイル中で一回書けばOKです

- 今回使うモジュールはどれもcsv限定ではなく，テキストに対して区切り文字を指定して読みこむものです．

※csvファイルはセルの区切り文字がカンマ，になっているテキストです．メモ帳でひらいてみると確認できます

次に知りたい②csvファイルの読み書き

1. 読み取り

空白セルがあるとき

```
import numpy as np
np.genfromtxt("sample.csv", delimiter="","", filling_values=0)
```

np.genfromtxt(“ファイル名.csv”, delimiter=“区切り文字”, filling_values=代わりの文字)

- filling_values：なんにもはいってないセルがあったときに代わりに詰めてもらう文字の指定．何も入れたくないときは””でいけます

列名ヘッダを飛ばしたりしたいとき

```
import numpy as np
np.loadtxt("sample.csv", delimiter="","", skiprows=1)
```

np.loadtxt(“ファイル名.csv”, delimiter=“区切り文字", skiprows=飛ばしたい行数)

2. 書き込み

```
np.savetxt("sample.csv", dataname,
delimiter=",")
```

np.savetxt(“ファイル名.csv", data名, delimiter=",")

- すでに存在しているのと同じファイル名を指定する場合，そのファイルを開いた状態で実行すると，エラーになるので注意．

次に知りたい③データフレーム

pandasのデータフレームはとても便利です．[**自主学習**]

- **データを行で取り出したい**
  データ名[始まり行番号:終わり行番号]
- **データの列を取り出したい**
  データ名[”列名”]
  複数列を取り出したいときは
  データ名[['列名1', '列名2']]
  ※かっこ[]が二重なのは，列名のリストを指定する形になっているから．
- **ある列が条件を満たす行だけ取り出す**
  データ名[“列名”] ==0 や >= 80 などとすると，
  その列の各行に対してTrue or Falseが返る．
  これを利用して，Trueなデータのみ抽出．
  データ名[データ名["列名"] ==0 とか >= 80とか]

```
data3 = data2[data2['A'] != ″不明″]
```

次に知りたい④NaNの判定法

- **欠損データNaNを含むデータを取り除く処理**

data4 = data3[data3['A'] == data3['A’]]

とすると，data3の各行のうち，列名AのセルがNaNでないもののみdata4に代入されます．

※どういうこと？

普通は同じものを比べたらTrueが返ります（0==0, “Hello” == “Hello”などの結果はTrue）．

しかし， PythonではNaNとNaNを比較する（NaN==NaN）とFalseが返る仕様になっているらしいです．

つまりある同一セルを比較した場合，NaNが入っていた場合のみFalseになります．

これを利用して，NaNでないものだけを取っています．

調査データ原本では欠損データが含まれることがよくあり，そのままでは推定が回りません．この処理をしておくとストレスなく推定に入れます．

データ編集コードの例

```
# dataedit
# パッケージpandasを使うのが便利
import pandas
# 書き込み先ファイル閉じないと回らないので注意
data2 = pandas.read_csv("data2.csv", encoding="utf_8")
# 文字コードのutf-8への変更は，メモ帳で開いて新規保存のときに文字コード指定が簡単
# print(csv)
data3 = data2[data2['A'] != "不明"]
data3 = data3[data3['A'] == data3['A'] ]
data3 = data3[data3['B'] == data3['B'] ]
data3 = data3[data3['C'] == data3['C'] ]
data3 = data3[data3['D'] == data3['D'] ]
data3 = data3[data3['D'] != "?" ]
print(data3)

# csvファイルとして出力
data3.to_csv("data3.csv")
```

（補）自主学習

- range()とprint()
- for文，if文，while文，try文
- データ型の判定
- 関数の定義の仕方
- pandasのデータフレームの使い方
- pandasのcsvファイルの扱い方
- 日付データの扱い方

練習問題

1. Welcome to BinN!とコンソール画面に表示
2. 1~100の数字をコンソール画面に表示
3. 1~100の数字をカンマで区切った一行でコンソール画面に表示(1, 2, …, 100)
4. 1~100の素数を表示

（発展）

緯度経度を変数に入力して，東京大学工学部一号館までの距離を算出してみてください.

これができたら，緯度経度が入力されたcsvファイルを読み込み，東京大学工学部一号館までの距離を算出し，csvファイルに書き込むコードを書いてみてください.

練習問題ヒント

1. print()
2. range()とfor文：PythonではRと違い，指定番号が0から始まるので注意.
3. print()の引数とif文・while文：print()ではそのままだと毎回改行されます．引数endを指定しましょう．また，100のあとにカンマを入れないためには，if文かwhile文を使うとよいでしょう.
4. if文とfor文：素数の求め方にはいろいろあります．まずもっとも実直なのは2-99の数で割り切れないものを素数と判断するもの．ネットで検索して効率のいいやり方もさがしてみてください.

（発展）

http://ikasumiwt.hatenadiary.jp/entry/2012/06/25/031357

緯度経度による距離算出はこのあたりを参考に．途中に書いてあるコードはPython対応じゃないので，Pythonでうごくように直しましょう.

第二部 最短経路探索法

Dijkstra, Bellman-Ford, A*

交通ネットワーク研究での大きな問題

計算量が多すぎる！

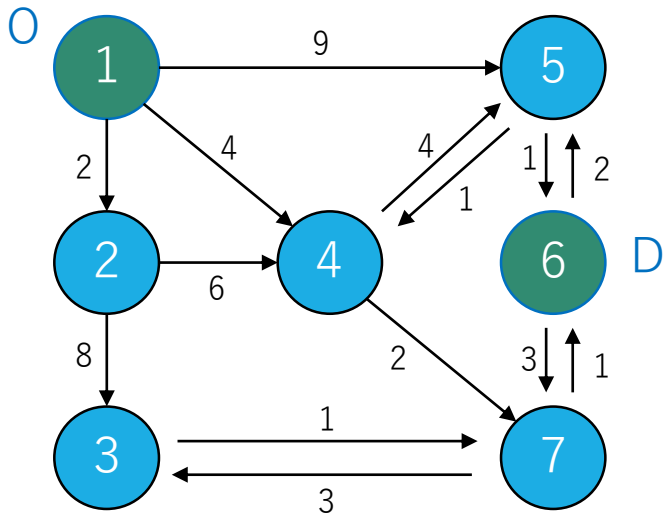
- 交通ネットワークはノード（点）とリンク（線）で記述することが多い.
- 実際のネットワークでは無数のノードとリンクの処理が必要.
- 経路選択では，リンクの組合せ問題になるので，膨大な量の計算をしなければならない！
- PCは人間の処理能力を大幅に超えていますが，それでも複雑な計算を大量にすることはスペック上難しいことがある.

→そこで，膨大な組みあわせの中から
必要なルートを効率よく列挙する必要があります

たとえば

- Origin: Node 1
- Destination: Node 6

このとき，最短経路は？

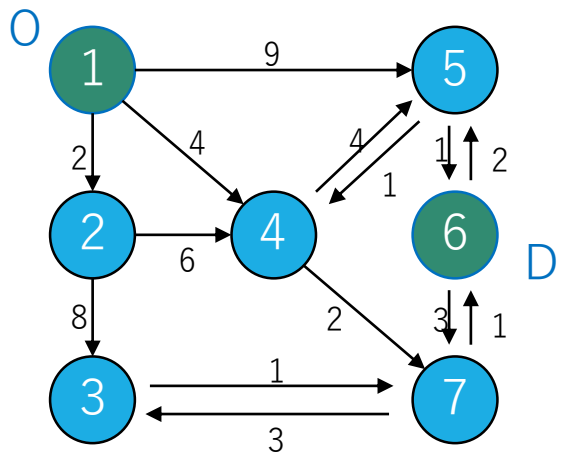


黒数字：リンクコスト

・距離
・運賃
・それらを重みづけしたもの
などさまざまなものを入れられます

答えは

- 答えは1→4→7→6



黒数字：リンクコスト

この答えを求めるべく全ての経路を書き出した場合

- 1 → 2 → 4 → 7 → 6 (11)
- 1 → 2 → 3 → 7 → 6 (14)
- 1 → 4 → 7 → 6 (7)
- 1 → 5 → 4 → 7 → 6 (16)
- 1 → 5 → 6 (10)

今回は列挙できましたが,
・全ノード数が4000個くらいあったら？
・Oが1以外のすべての場合についても
　計算する必要があったら？
…**計算量が，多すぎる！**

→最短経路探索法

様々なアルゴリズムがあります

- **Dijkstra法**
  ：最も一般的な最短経路探索アルゴリズム.
  各辺の重みが非負でないと使えない.
  起点からすべての終点までの最短経路と最小費用が同時に求まる.
- **Bellman-Ford法**
  ：負の重みのリンクを含むグラフにおいても最短経路を求めることができるアルゴリズム．基本的にはダイクストラ法より遅い.
- **A＊**
  ：ダイクストラ法の改良アルゴリズム.
  終点までの距離の推定値（ヒューリスティック関数）を利用する.
  基本的にダイクストラ法よりも高速だが，ダイクストラ法と異なり，**一回に計算できるのは起終点1ペアに対する最短経路のみ**.

Dijkstra法

「最短とわかっているところから確定していく」

すべてのノードは、既に最短経路が求まっているノードの集合Kと

まだ最短経路が求まっていないノードの集合$\overline{K}$のどちらかに分類できる

各ノードiについて以下の変数を定めます.

・ c_i：始点からノードiまでの最小交通費用

$i \in K$のとき（確定済）　c_i:最小交通費用

$i \in \overline{K}$のとき（未確定）　c_i:部分的最小費用…ここまでの最小費用

・ t_{ij}：ノードiからノードjまでの交通費用

・F_i：ポインタ（最短経路を列挙する際に使う．後述）

アルゴリズム

Step1
起点と初期値の設定

全てのノード$\{j\}$について，
部分的最小費用$c_j = \infty, F_j = 0, j \in \overline{K}$とする．
起点をoとし，$c_o = 0, i = o$とする．ノードoを集合Kに移す

Step2
部分的最小交通費用の更新

ノードiから出る全リンクの終点ノード$\{m\}$について，
$c_m > c_i + t_{im}$ならば，$c_m = c_i + t_{im}$に更新し，$F_m = i$ とする

Step3
最小交通費用の確定

集合$\overline{K}$に属すすべてのノードの中での
部分的最小費用が最小となっているノードを求め，
これをノードjとする．ノードjを集合Kに移す
$c_j = \min_p(c_p)\ (p \in \overline{K})$

Step4
不確定ノードのチェック

$c_p = \infty$以外のすべてのノードが集合Kに移されているかチェック

NO

$i = j$ として Step2へ

YES

終了

Step5
経路の導出

（おさらい）

K：既に最短経路が求まっているノードの集合

$\overline{K}$：まだ最短経路が求まっていないノードの集合

・c_i：始点からノードiまでの最小交通費用

$i \in K$のとき（確定済）　c_i: 最小交通費用

$i \in \overline{K}$のとき（未確定）　c_i:部分的最小費用

・t_{ij}：ノードiからノードjまでの交通費用

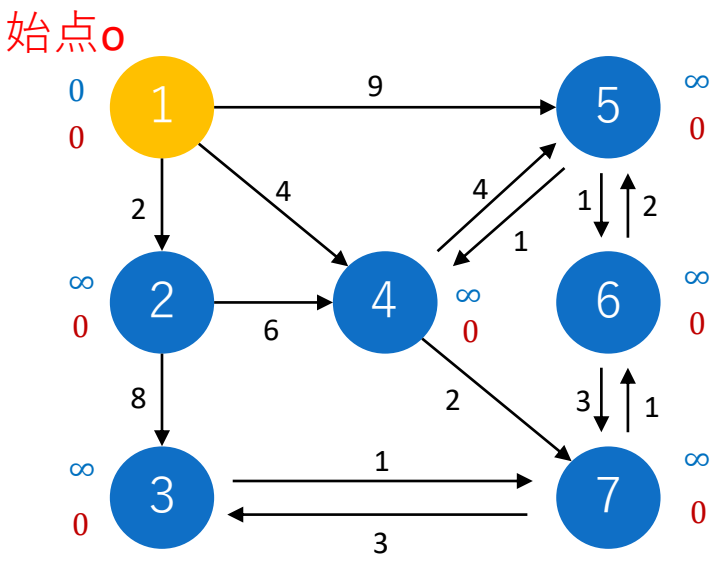
・F_i：ポインタ

Step1 起点と初期値の設定

全てのノード$\{j\}$について，
部分的最小費用$c_j = \infty, F_j = 0, j \in \overline{K}$とする．
起点をoとし，$c_o = 0, i = o$とする．ノードoを集合Kに移す

<Step1>

$i = 1$



（おさらい）
- K：既に最短経路が求まっているノードの集合
- $\overline{K}$：まだ最短経路が求まっていないノードの集合
- c_i：始点からノードiまでの最小交通費用
  - $i \in K$のとき（確定済）　c_i:最小交通費用
  - $i \in \overline{K}$のとき（未確定）　c_i:部分的最小費用
- t_{ij}：ノードiからノードjまでの交通費用
- F_i：ポインタ

黒数字：リンクコスト
赤数字：F_m（ポインタ）
青数字：c_m（最小費用）

c_m							F_m						
1	2	3	4	5	6	7	1	2	3	4	5	6	7
0	∞	∞	∞	∞	∞	∞	0	0	0	0	0	0	0

K	1
$\overline{K}$	2,3,4,5,6,7

Step2 部分的最小交通費用の更新

ノードiから出る全リンクの終点ノード$\{m\}$について，
$c_m > c_i + t_{im}$ならば，$c_m = c_i + t_{im}$に更新し，$F_m = i$ とする

ノード 1 から行けるのはノード2, 4, 5→ $m = 2,4,5$

$$c_2' = c_1 + t_{12} = 0 + 2 = 2 < \infty (= c_2)$$
$$c_4' = c_1 + t_{14} = 0 + 4 = 4 < \infty (= c_4)$$
$$c_5' = c_1 + t_{15} = 0 + 9 = 9 < \infty (= c_5)$$

$m = 2,4,5$すべてで$c_m' < c_m$なので，すべて更新

また，$F_m = i = 1$とする

（おさらい）

K：既に最短経路が求まっているノードの集合

$\overline{K}$：まだ最短経路が求まっていないノードの集合

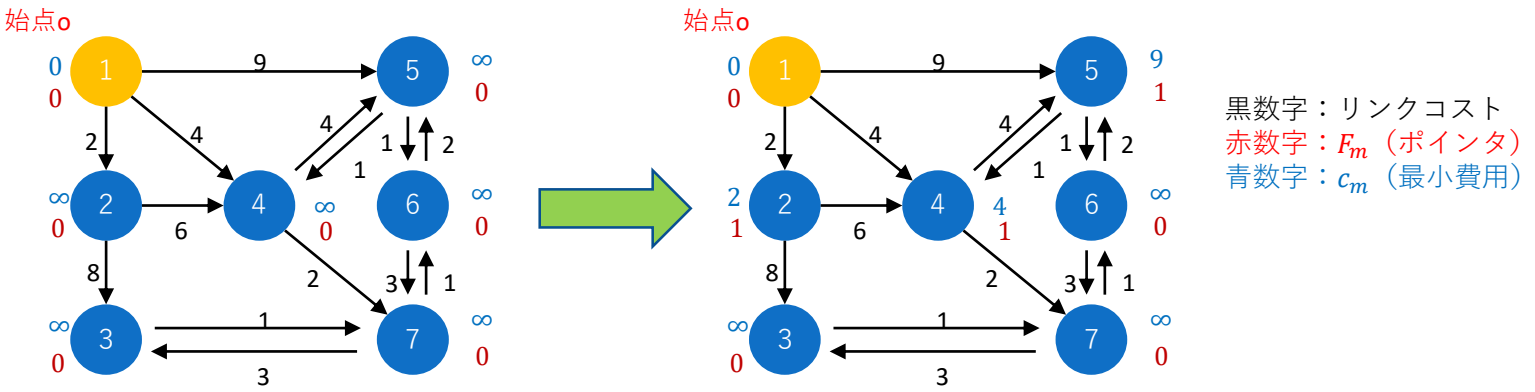
・c_i：始点からノードiまでの最小交通費用

　$i \in K$のとき（確定済）　c_i:最小交通費用

　$i \in \overline{K}$のとき（未確定）　c_i:部分的最小費用

・t_{ij}：ノードiからノードjまでの交通費用

・F_i：ポインタ



黒数字：リンクコスト
赤数字：F_m（ポインタ）
青数字：c_m（最小費用）

Step3 最小交通費用の確定

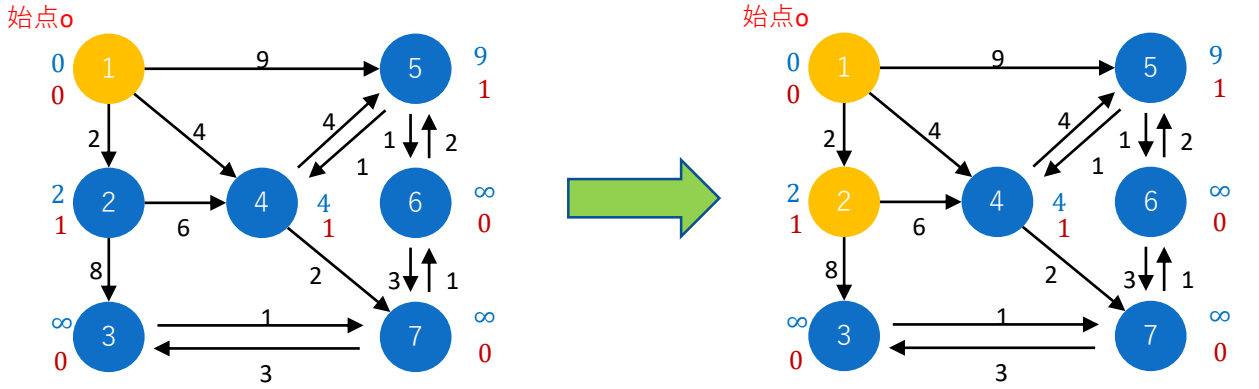
集合$\overline{K}$に属すすべてのノードの中での
部分的最小費用が最小となっているノードを求め，
これをノードjとする．ノードjを集合Kに移す

$$c_j = \min_p (c_p)\ (p \in \overline{K})$$

$\overline{K} = \{2,3,4,5,6,7\}$に属すノードの中で費用が最小となるのは、ノード２

$\rightarrow j = 2$ として、ノード2を集合Kへ移す

ノード2を含む最短経路：$1 \rightarrow 2$　最小交通費用：$c_2 = c_1 + t_{12} = 0 + 2 = 2$



（おさらい）

K：既に最短経路が求まっているノードの集合

$\overline{K}$：まだ最短経路が求まっていないノードの集合

・c_i：始点からノードiまでの最小交通費用

　$i \in K$のとき（確定済）　c_i: 最小交通費用

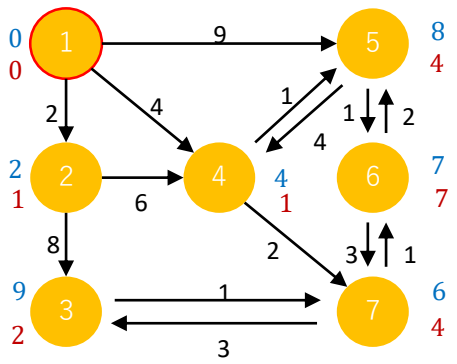
　$i \in \overline{K}$のとき（未確定）　c_i:部分的最小費用

・t_{ij}：ノードiからノードjまでの交通費用

・F_i：ポインタ

Step4 不確定ノードのチェック

$c_p = \infty$以外のすべてのノードが
集合Kに移されているかチェック



$c_p \neq \infty$かつ$p \in \overline{K}$なるpが残っている

→次は$i = 2$でstep2～step4を行う

→ $i = 4$→ $i = 7$→ $i = 6$→ $i = 5$→終了

Step5 最短経路の導出

ポインタF_mを目印に遡っていく

・ノード 6 までの最短経路

$$6 \rightarrow (F_6 =)7 \rightarrow (F_7 =)4 \rightarrow (F_4 =)1$$

（おさらい）

K：既に最短経路が求まっているノードの集合

$\overline{K}$：まだ最短経路が求まっていないノードの集合

・c_i：始点からノードiまでの最小交通費用

　$i \in K$のとき（確定済）　c_i:最小交通費用

　$i \in \overline{K}$のとき（未確定）　c_i:部分的最小費用

・t_{ij}：ノードiからノードjまでの交通費用

・F_i：ポインタ

サンプルコードを見てみよう

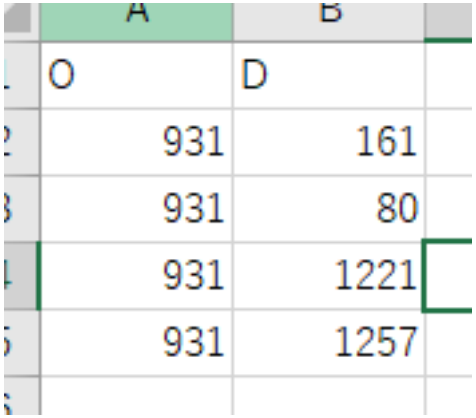
みなさんにお渡しするデータ・コード類は以下

プログラム

- adj_matrix.py：ネットワークデータから隣接行列を作成するコード
- dijkstra.py：ダイクストラ法による最短経路探索コード

データ

- ネットワークデータ（渋谷/松山）：node.csvとlink.csvで１セット
- ODlist.csv：探索したい経路の起終点ノードを格納したcsvファイル



	A	B
	O	D
2	931	161
3	931	80
4	931	1221
5	931	1257
6		

Pythonによる最短経路探索①

インプットデータ（渋谷）

- Shibuya_node.csv：ノード（交差点など）の位置座標
- Shibuya_link.csv：リンク（道）の起終点情報とリンクコスト

ノード番号　緯度　経度

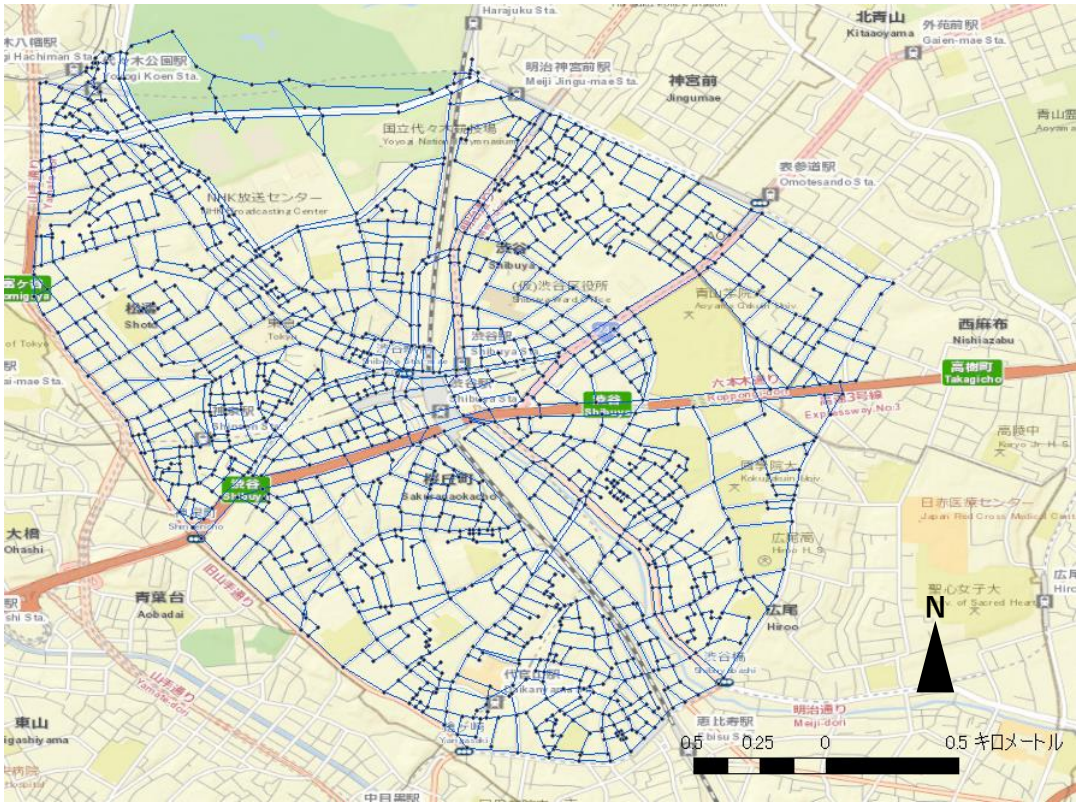
nodeID	latitude	longitude
1	35.66447	139.6913
2	35.66438	139.6922
3	35.66429	139.6927
4	35.66413	139.6939
5	35.66446	139.695
6	35.66392	139.6954

Shibuya_node.csv

リンク番号　起点　終点　リンクコスト（今回は距離(m)）

LinkID	n1	n2	LinkCost
0	1	2	78.28223
1	1	11	109.1107
2	1	1521	28.12929
3	2	1	78.28223
4	2	3	52.72191
5	2	7	37.86279
6	3	2	52.72191

Shibuya_link.csv



Pythonによる最短経路探索②

- （ラベル付き）隣接行列を作成する：**adj_matrix.py**

adj_matrix.py

```python
import pandas as pd
import numpy as np

#####################################################
###データ読み込み######
#####################################################

link = pd.read_csv('input/Shibuya_link.csv') # リンクデータをインプット
node = pd.read_csv('input/Shibuya_node.csv') # ノードデータをインプット
# print(link.head())
# print(node.head())
nl = len(link)
nn = max(node['nodeID'])
print(nl) # リンク数
print(nn) # ノード数

#####################################################
###実行######
#####################################################

# 「型」となる零行列を用意します（ノード数×ノード数）.
mat = np.zeros([nn,nn])

# 各列の読み込み
olist = link['n1'] #起点リスト
dlist = link['n2'] #終点リスト
clist = link['LinkCost'] #リンクコストリスト

# 一行ずつ読み込んで零行列の要素を更新
for i in range(nl):
    onode = olist[i]
    dnode = dlist[i]
    mat[onode-1,dnode-1] = clist[i]


#####################################################
###出力######
#####################################################
np.savetxt('input/Shibuya_matrix.csv', mat, delimiter=',')
```

←パッケージの読み込み

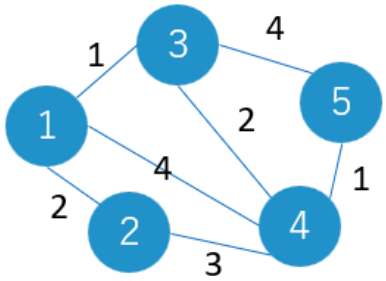
↑リンク・ノードデータ入力

↓（ラベル付き）隣接行列出力

隣接行列：ノードの接続関係を行列で表したもの

ラベル付き隣接行列：隣接を表す1の代わりに各ルートに対応する費用（コスト）を用いたもの

- 隣接している（経路がある）：1
- 隣接していない（経路がない）：0



隣接行列

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

ラベル付き隣接行列

$$\begin{bmatrix} 0 & 2 & 1 & 3 & 0 \\ 2 & 0 & 0 & 3 & 0 \\ 1 & 0 & 0 & 2 & 4 \\ 3 & 3 & 2 & 0 & 1 \\ 0 & 0 & 4 & 1 & 0 \end{bmatrix}$$

Pythonによる最短経路探索③

- Dijkstra法による最短経路探索：**dijkstra.py**

```
#################################################
###パッケージの読み込み#######
#################################################

import numpy as np
from scipy.stats import rankdata
#from numpy import *
import pandas as pd
import itertools
import math

#################################################
###データ読み込み#######
#################################################

# ラベル付き隣接行列: 手入力はこちら！
"""
m = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
]
"""
# csvに書き込んでもよいです 量が多いときはこれ自体をコード書いて処理してつくるのがよいでしょう
m = np.genfromtxt("input/Shibuya_matrix.csv", delimiter=",", filling_values=(0))

# ノード数は隣接行列の要素数です
m_node = len(m)
print(m_node)

OD = pd.DataFrame(np.loadtxt("input/ODlist2.csv", delimiter=",",skiprows=1))
OD.columns = ["O", "D"]

Olist = OD['O']
Dlist = OD['D']
```

データの読み込み
：隣接行列とOD表

```
# ネットワークmにおいて出発地oriから目的地desまでの最短経路を探していきます
def search(m, ori, des):
    # 初期設定
    max_cost = float('inf')  # 初期コストは無限大
    unchecked = [False] * m_node  # 未確定ノードの集合
    cost = [max_cost] * m_node  # 起点から各ノードへの最小費用
    f_prev = [None] * m_node  # 最短経路を列挙するための配列
    cost[ori-1] = 0  # 起点のノードの距離は0とする
    f_prev[ori-1] = ori  # 起点前のノードは起点とする
    now = ori  # 現在地を起点とする

    while True:
        min = max_cost  # 変数minは現段階の最小費用を表す（初期は無限大）
        next = -1  # nextは起点から最小費用のあるノードを表す(初期－1)
        unchecked[now-1] = True
        for i in range(m_node):  # 変数iはノード（0~4）
            if unchecked[i]: continue  # ノードが確定していない場合ループが続く
            if m[now-1][i]:  # 今のノードiと起点(now=ori)が接続しているかどうか
                tmp_cost = m[now-1][i] + cost[now-1]  # 一時的の費用を計算し、最小費用の更新
                if cost[i] > tmp_cost:
                    cost[i] = tmp_cost
                    f_prev[i] = now  # 直前のノードに更新
            if min > cost[i]:  # 現段階の最小費用と最小費用を持つノードを更新
                min = cost[i]
                next = i+1
        now = next  # 確定された最小費用持つノードが新しい”起点”となる
        if next == -1: break  # ノード番号が－1になるまで(すべてノードが確認される)

    #print_path(f_prev, cost) # 各接続しているノード間ルートの費用
    #print(get_path(ori, des, f_prev), cost[des])
    return [get_path(ori, des, f_prev), cost[des-1]]


# 結果の出力
def print_path(f_prev, cost):
    for i in range(len(f_prev)):
        print("%d, prev = %d, cost = %d" % (i, f_prev[i], cost[i]))


def get_path(ori, des, f_prev):
    path = []
    now = des
    path.append(now)  # pathの最後にnowを加えている
    while True:
        path.append(f_prev[now-1])  # f_prevには一つ前のノードが入ってる。
        if f_prev[now-1] == ori: break  # 格納されたノードを逆にたどっていけば経路が求められる
        now = f_prev[now-1]
    path.reverse()
    return path
```

関数の定義
：リストの扱い
に注意すること

Pythonによる最短経路探索④

実行する

```
##################################################
###実行#######
##################################################

for v in range(len(Olist)):

    origin = int(Olist[v])
    destination = int(Dlist[v])

    print(search(m,origin,destination))
```

▼ **Run**

ノード数

```
1612
[[931, 186, 183, 177, 176, 175, 174, 170, 168, 167, 163, 162, 161], 426.90161687700004]
[[931, 186, 943, 944, 178, 172, 171, 958, 166, 154, 202, 153, 152, 849, 151, 150, 136, 137, 138, 93, 90, 87, 83, 81, 80], 1371.50684226]
[[931, 1233, 1147, 294, 295, 297, 300, 350, 353, 354, 355, 356, 357, 358, 359, 363, 1066, 1115, 392, 399, 420, 429, 759, 1142, 1448, 1447, 760], 1415.9943885799998]
[[931, 1039, 539, 924, 512, 1207, 1042, 507, 1209, 505, 504, 503, 689, 659, 660, 661, 685, 686, 687, 688, 1239, 1240, 1250, 1251, 1257], 1803.7652265199995]
4.435364723205566

Process finished with exit code 0
```

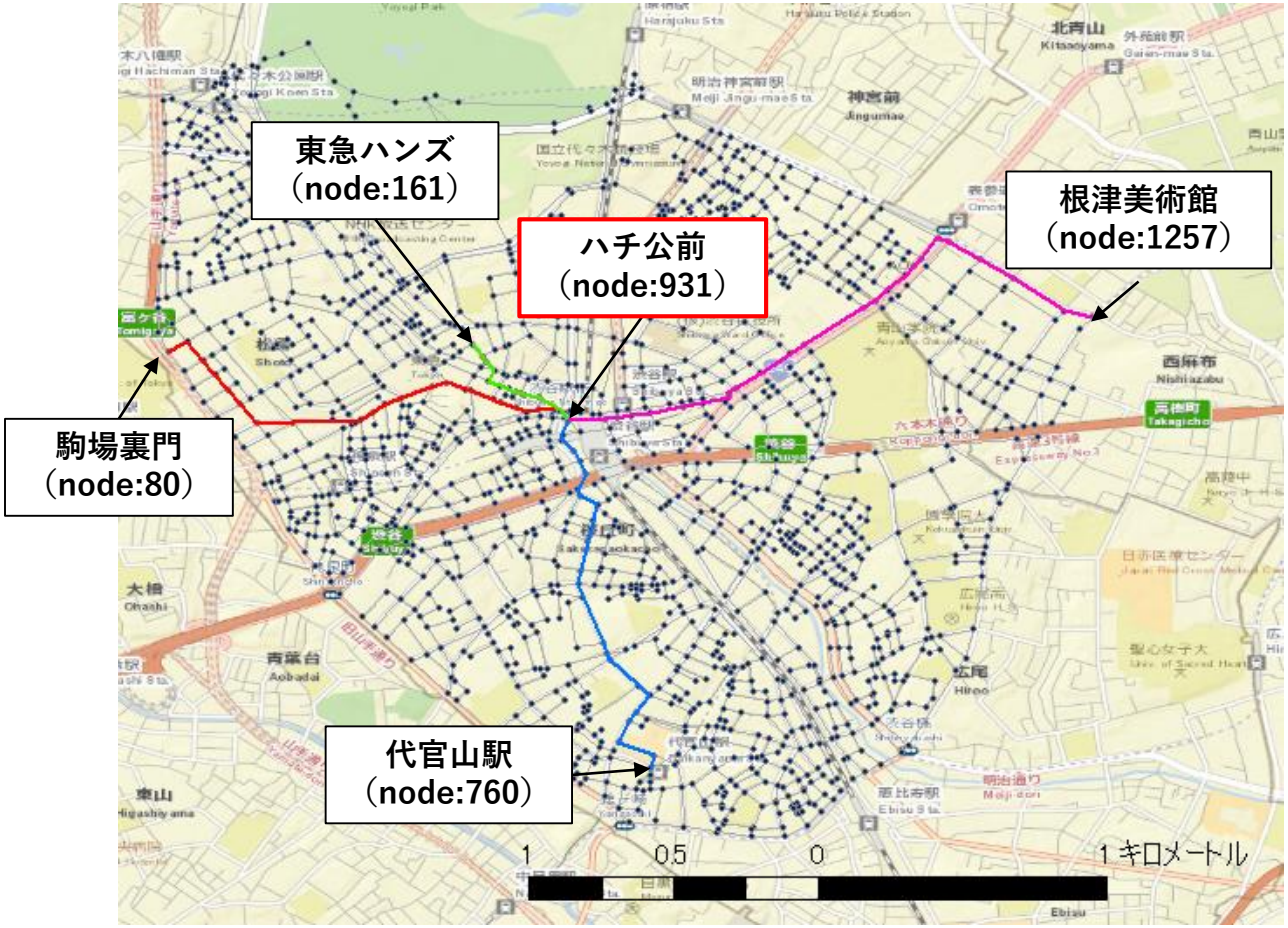
経路情報

計算時間
（timeモジュール）

最短経路が分かった！

Pythonによる最短経路探索⑤

GISに落としてみるとこんな感じ



課題

課題

課題：最短経路アルゴリズムの実装

Step 1:

- Dijkstra法について理解したうえで，配布したコードを解読してください．
- 渋谷か松山のネットワークデータを用いて，Dijkstra法による最短経路探索を実行し，結果をGISで表示してください．

Step 2:

- Bellman-Ford法かA*アルゴリズムについて各自ひとつ調べてまとめてください．(http://bin.t.u-tokyo.ac.jp/tansaku_18/を参考にしてください)
- 調べた最短経路探索手法をPythonで実装してください．
- Step 1で実行したものと比較して，計算時間の比較をしてみましょう．

課題

参考

- 最初にイメージをつかむならこれ
  https://www.youtube.com/watch?v=niypzttxyLE&list=PLDb67L4FP3uhhCwUlJzkAdA4eUCB3Si-c
- A*アルゴリズムについて
  元論文：https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4082128
  ウィキペディアの説明が結構わかりやすい：https://ja.wikipedia.org/wiki/A*
- Bellman-Ford法について
  元論文：https://www.ams.org/journals/qam/1958-16-01/S0033-569X-1958-0102435-2/S0033-569X-1958-0102435-2.pdf・
  わかりやすい記事：
  https://nw.tsuda.ac.jp/lec/BellmanFord/
  https://qiita.com/wakimiko/items/69b86627bea0e8fe29d5

Appendix

ヒープ構造

$c_j = \min_p(c_p)\ (p \in \overline{K})$の求め方：ヒープ構造

ダイクストラ法の中のStep3の「最小」の求め方は？

Step3 …部分的最小費用が最小となっているノードを求め… $c_j = \min_p(c_p)\ (p \in \overline{K})$

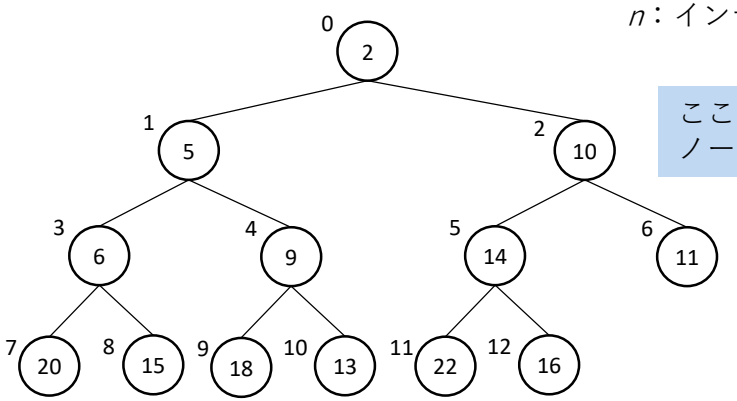
ヒープ構造

- 最小値を効率的に求めるために用いるデータ構造

「**ヒープ条件**」を満たすように配列（〇つき数字）を並べ，インデックスをつける

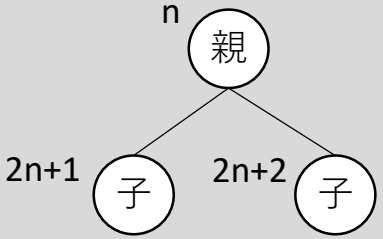
$$B(n) \geq B(2n+1), B(2n+2)$$

n：インデックス，*B(n)*：配列



ここで〇の中に入っている数字はノード番号ではなく，**配列（値）**

必ず子供よりも親が小さい構造を作る

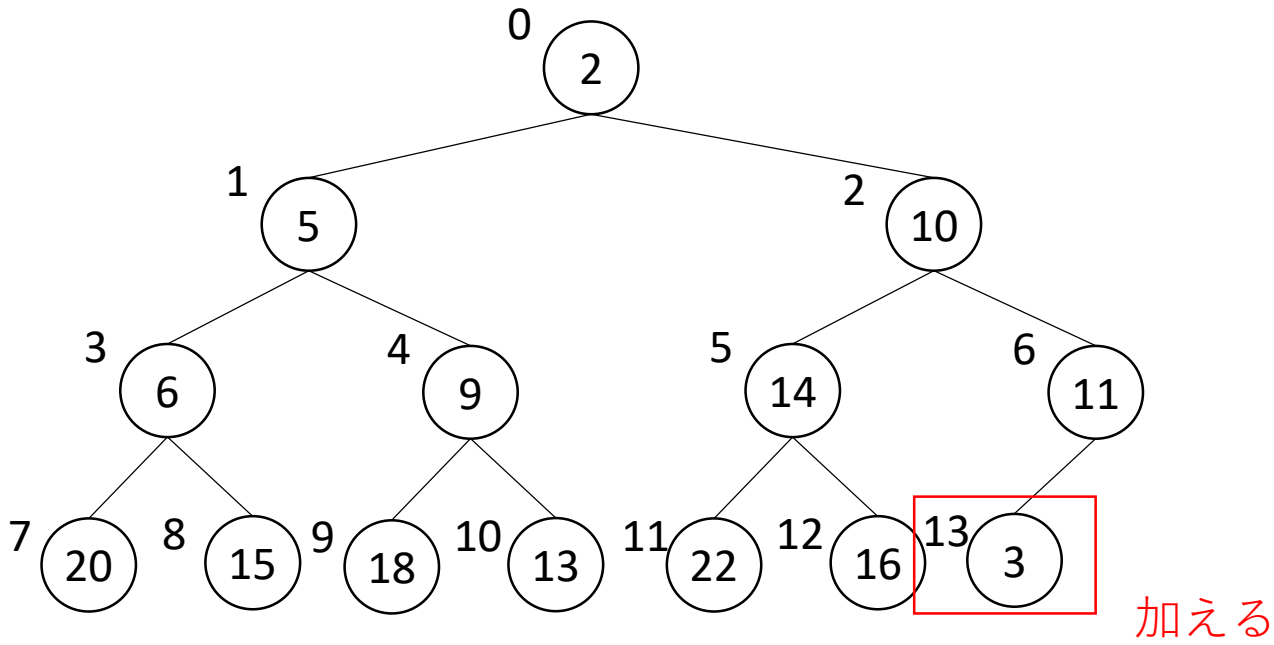


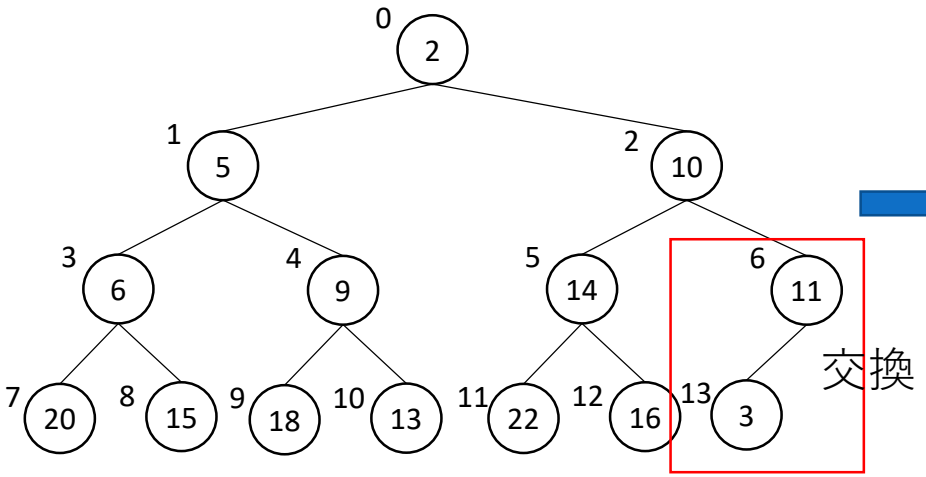
→最小のものは一番上に来る

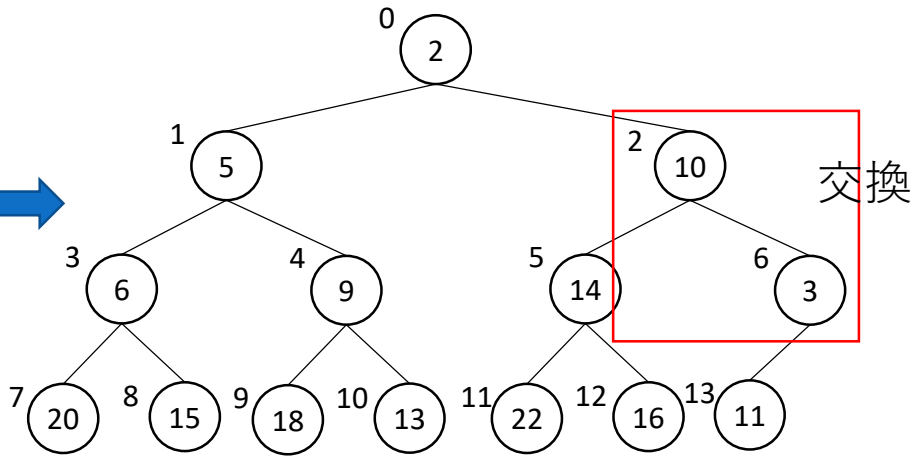
n	0	1	2	3	4	5	6	7	8	9	10	11	12
B(n)	2	5	10	6	9	14	11	20	15	18	13	22	16

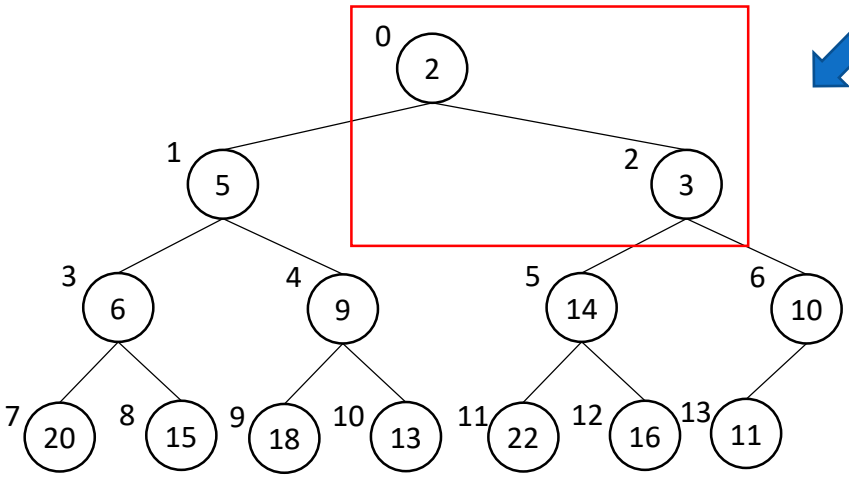
数字追加のとき

インデックスn	0	1	2	3	4	5	6	7	8	9	10	11	12	13
配列B(n)	2	5	10	6	9	14	11	20	15	18	13	22	16	3



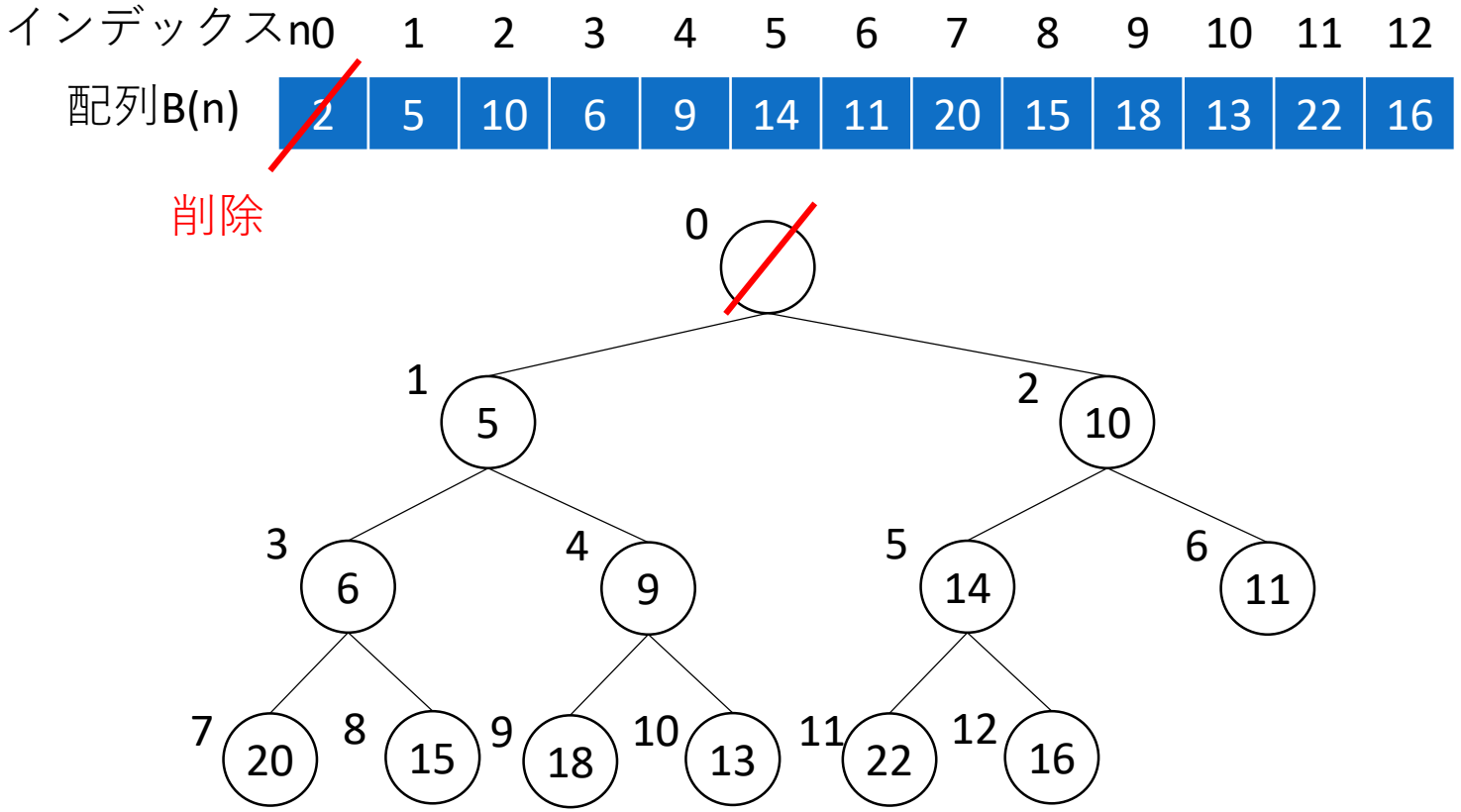


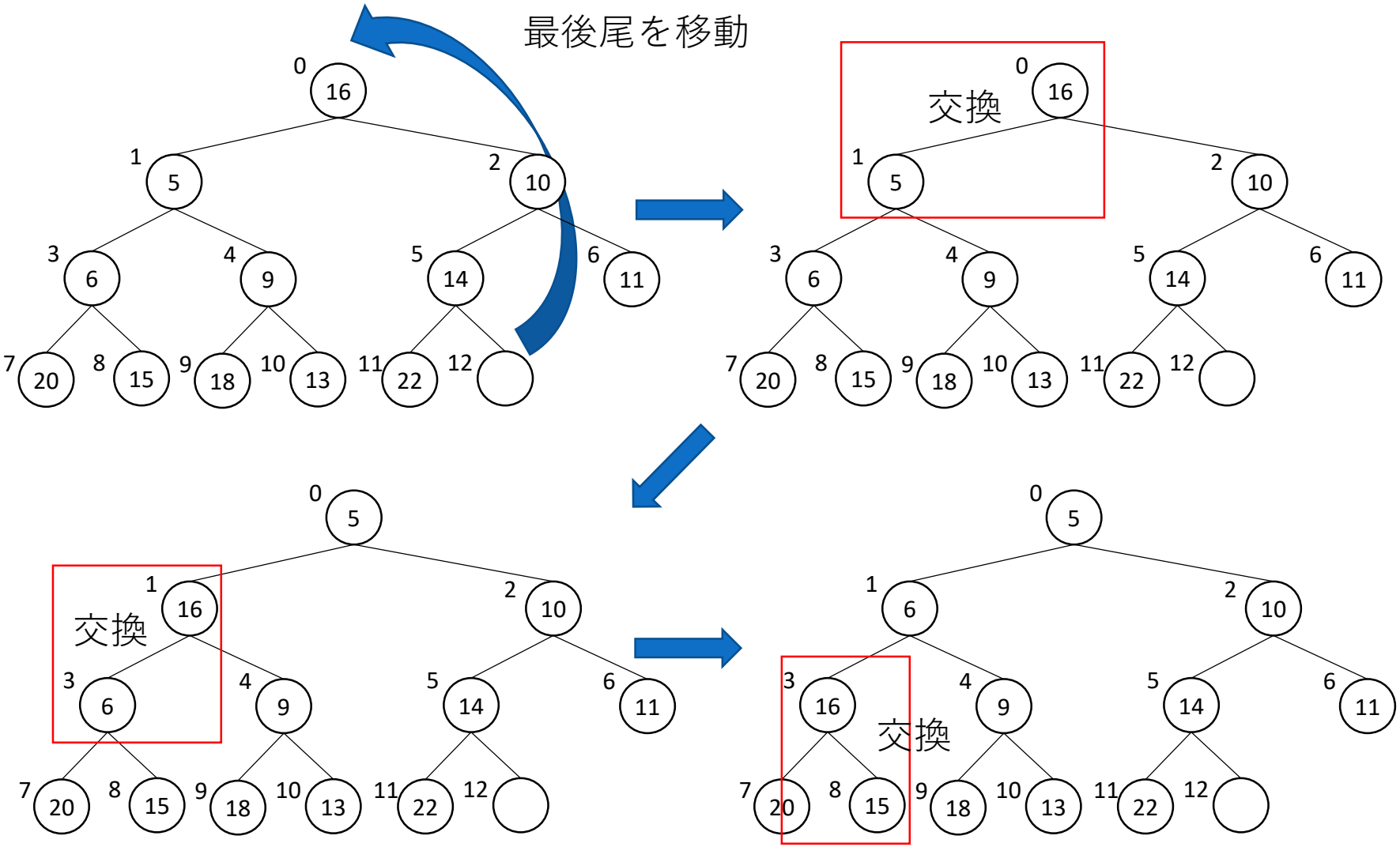


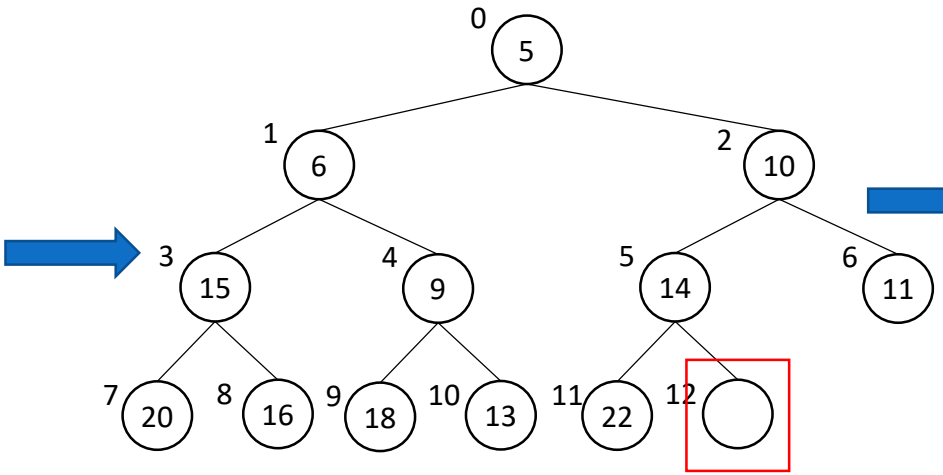


ヒープ条件を満たしているので完了

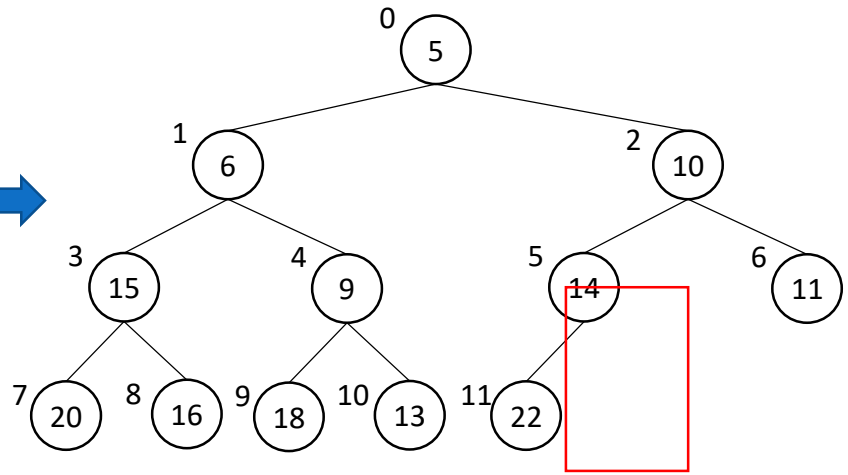
数字削除のとき


インデックスn0 1 2 3 4 5 6 7 8 9 10 11 12
配列B(n)
2 5 10 6 9 14 11 20 15 18 13 22 16
削除
0
1
5
2
10
3
6
4
9
5
14
6
11
7
20
8
15
9
18
10
13
11
22
12
16


最後尾を移動
交換
交換
交換
0 16
1 5
2 10
3 6
4 9
5 14
6 11
7 20
8 15
9 18
10 13
11 22
12



削除



ヒープ条件を満たしているので完了