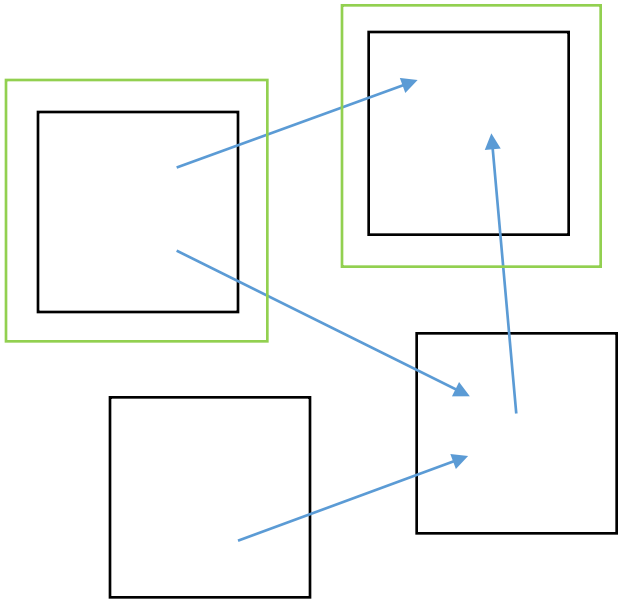
Intro to Computer Science

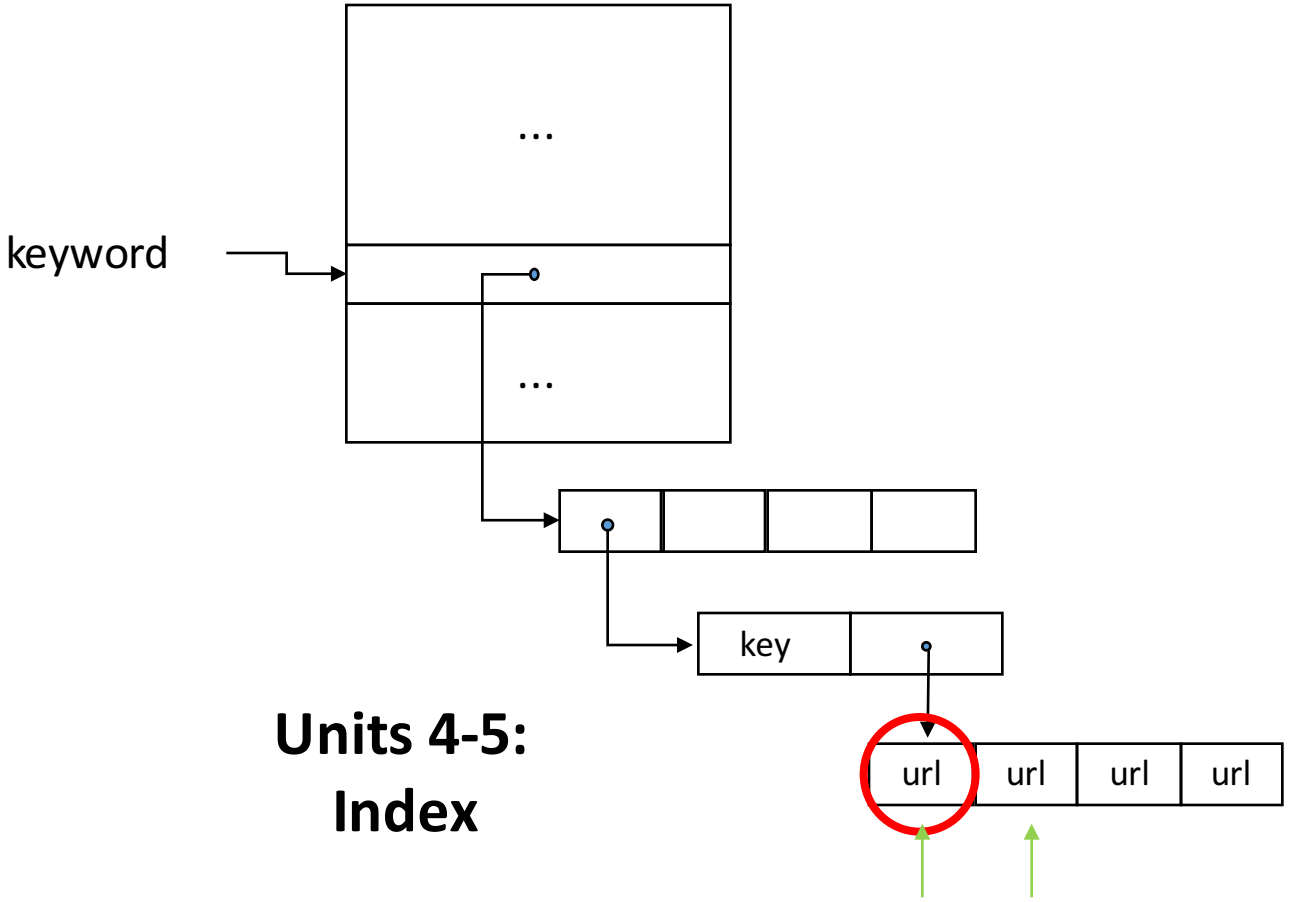
Local Laboratory

*** Udacity – Intro to Computer Science**

Unit 6: Ranking Web Pages

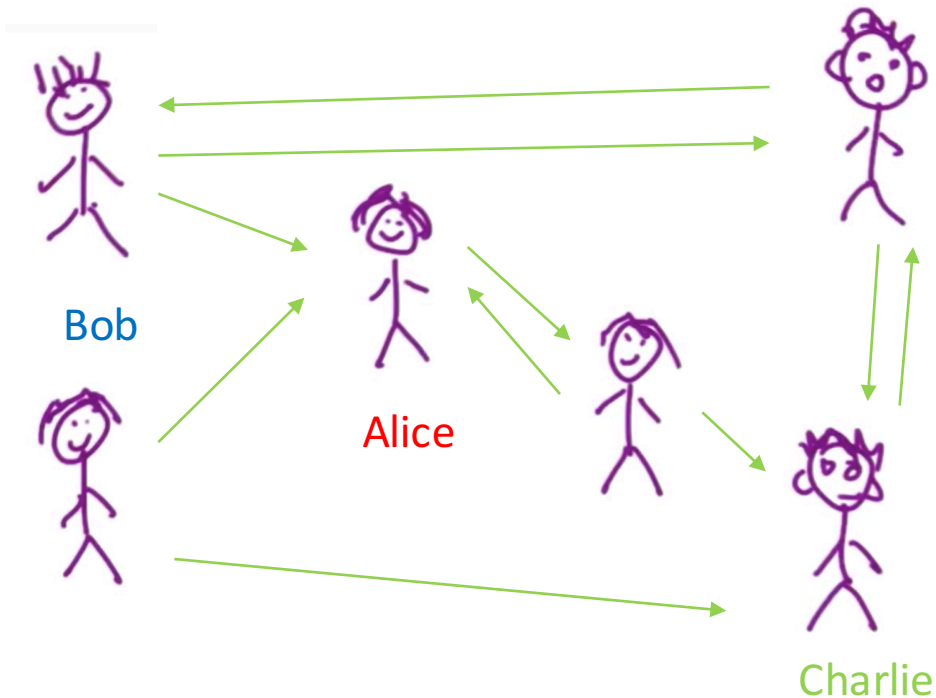


Units 1-3: Crawler



Units 4-5: Index

Popularity



popularity = 3

popularity(p) = number of people who are friends with p

$$popularity(p) = \sum_{f \in friends\ of\ p} popularity(f)$$

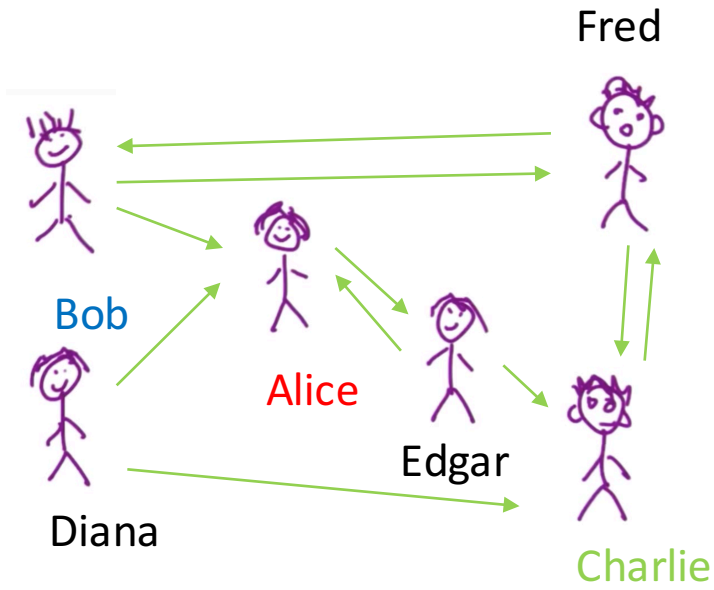
```
def popularity(p):
    score = 0
    for f in friends(p):
        score = score + popularity(f)
    return score
```

Quiz: Good Definitions

```
def popularity(p):
    score = 0
    for f in friends(p):
        score = score + popularity(f)
    return score
```

No base case!

circular definition!



위의 정의는 좋은 정의인가? Recursive Definition

❑ yes ❑ no

Recursive Definitions

Two parts:

1. Base case – a starting point
   - Not defined in terms of itself
   - Smallest input – already know the answer

2. Recursive case
   - Defined in terms of “smaller” version of itself

Defining Procedures Recursively

factorial(n) = n * (n-1) * (n-2) * ... * 1

factorial(n-1)

factorial(0) = 1 Base Case

factorial(n) = n * factorial(n-1) Recursive Case

n > 0

Quiz: Recursive Factorial

0 또는 자연수를 입력받아서, 입력한 숫자에 대한 팩토리얼 값을 계산하여 리턴하는 factorial이라는 프로시져를 정의하시오.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

factorial(0) = 1
factorial(n) = n * factorial(n-1)

factorial(4) ⟶ 24
4 * factorial(3) ⟶ 6
3 * factorial(2) ⟶ 2
2 * factorial(1) ⟶ 1
1 * factorial(0)
1

Quiz: Circular Definitions

factorial(0)
-> 1

```
def popularity(p):
    score = 0
    for f in friends(p):
        score = score + popularity(f)
    return score
```

No base case!

circular definition!

popularity('Alice') = 1

$$popularity(p) = \sum_{f \in friends\ of\ p} popularity(f)$$

```
def popularity(p):
    score = 0
    for f in friends(p):
        score = score + popularity(f)
    return score
```

```
if p == 'Alice':
    return 1
```

왼쪽과 같은 정의는 잘 동작하겠는가?

- ❑ Only if everyone is friends with 'Alice'.
- ❑ Only if no one is friends with 'Alice'.
- ❑ Only if there is a friendship path from everyone to 'Alice'.
- ❑ Only if there are no cylces in the graph.
- ❑ No.

Quiz: Relaxation

Relaxation Algorithm

-> start with a guess
while not done:
make the guess better

Base case

popularity(0 , p) -> score (1)

time step: 0 — person: p

popularity(t , p) -> $\sum_{f \in friends\ of\ p} popularity(t-1, f)$

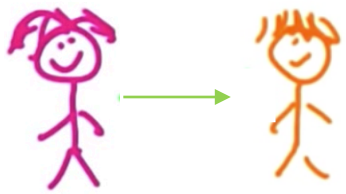
```
def popularity(t, p):
    if t == 0:
        return 1
    else:
        score = 0
        for f in friends(p):
            score = score + popularity(t-1, f)
        return score
```

위의 정의는 좋은 recursive definition인가?

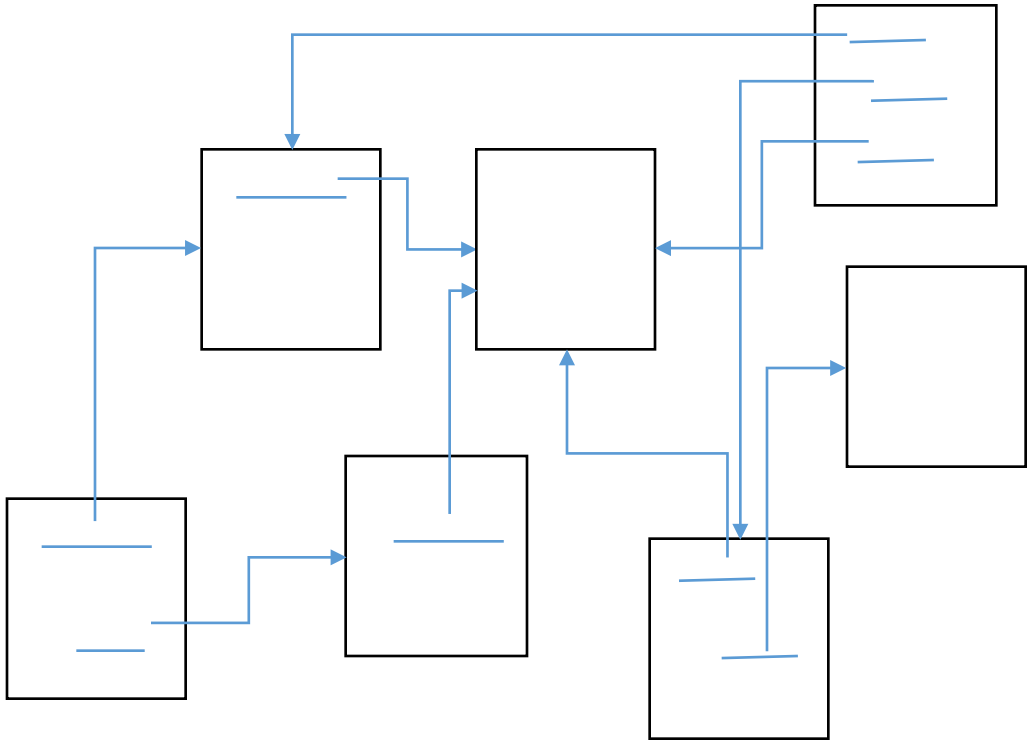
- ❑ Yes
- ❑ Only if people can't be friend themselves.
- ❑ Only if everyone has at least one friend.
- ❑ Only if everyone is more popular than 'Alice'

Page Rank

Ranking Web Pages



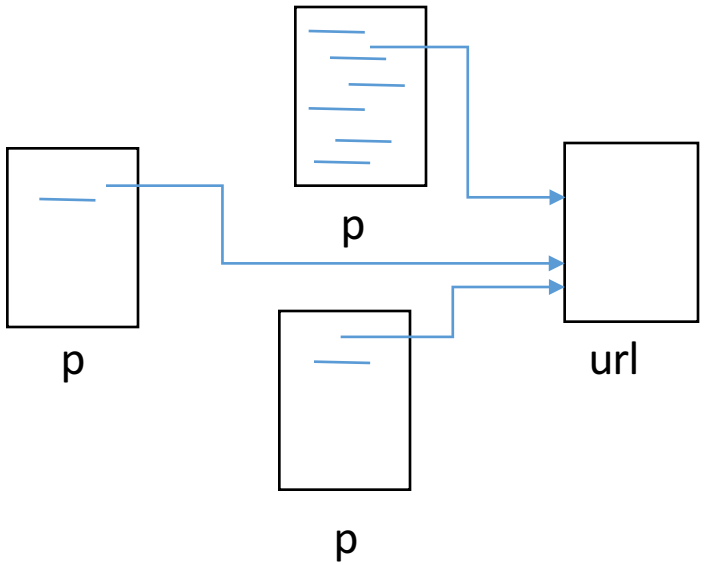
Freinds



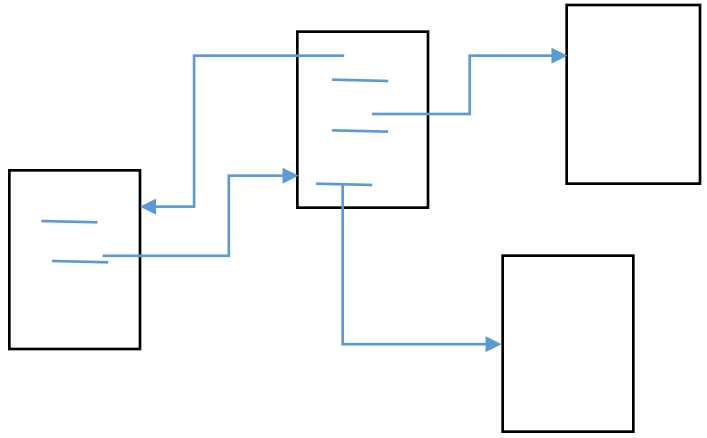
Page Rank

rank(0, url) -> 1

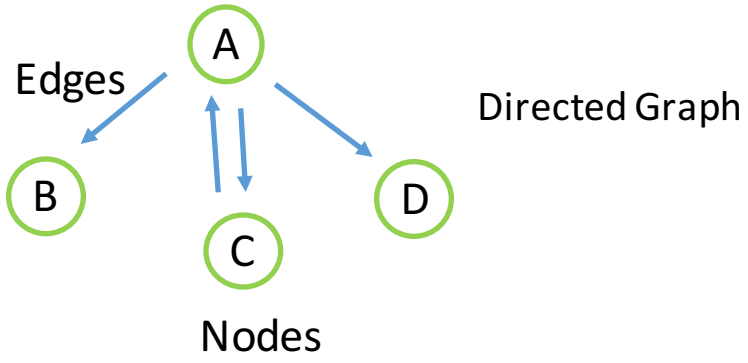
rank(t, url) -> $\sum_{p \in inlinks[url]}$ rank(t-1, p) **/ outlinks[p]**



~~Page Rank~~ **Local Rank**



Graph



{ url: [pages it links to], ... }

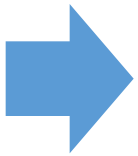
{ ‘A’: [‘B’, ‘C’, ‘D’],
‘B’: [ ],
‘C’: [‘A’],
‘D’: [ ] }

Local Rank

Building the Link Graph

crawl_web(seed) -> index , graph

```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    index = {}
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            union(tocrawl, get_all_links(content))
            crawled.append(page)
    return index
```



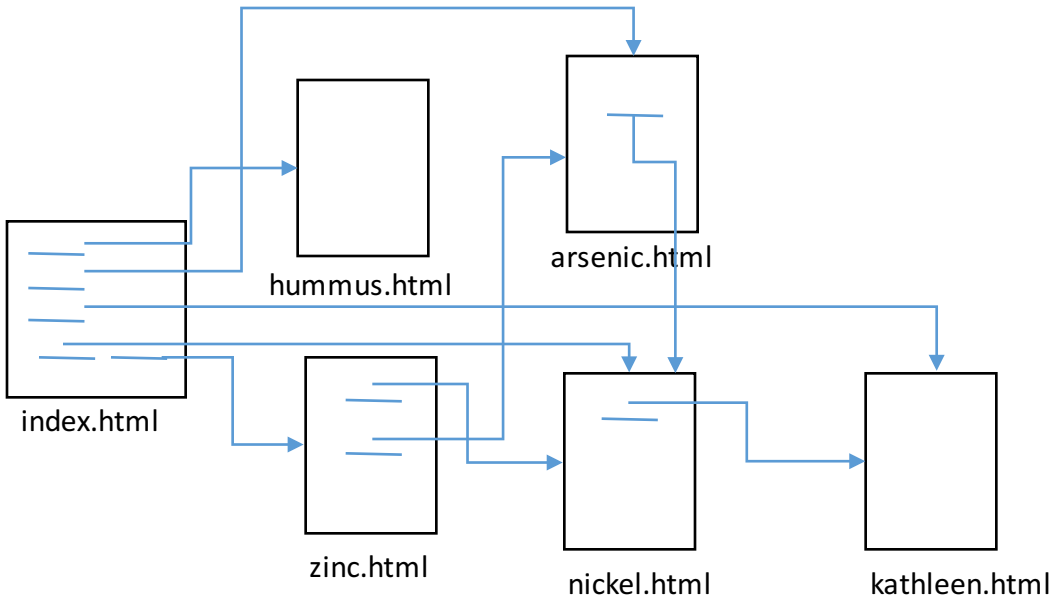
```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    index = {}
    graph = {}
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            outlinks = get_all_links(content)

            union(tocrawl, outlinks)
            crawled.append(page)

    return index, graph
```

Quiz: Implementing Local Rank

crawl_web 프로시저를 기존에 index만 리턴하는 것 대신 index와 graph를 리턴하도록 수정하시오.
그래프는 아래와 같은 entry들을 가지는 Dictionary 타입이어야 한다.

url: [url, url, url]

page pages that link to target



Implementing Local Rank

code

```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    index = {}
    graph = {}
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            outlinks = get_all_links(content)
            graph[page] = outlinks
            union(tocrawl, outlinks)
            crawled.append(page)

    return index, graph
```

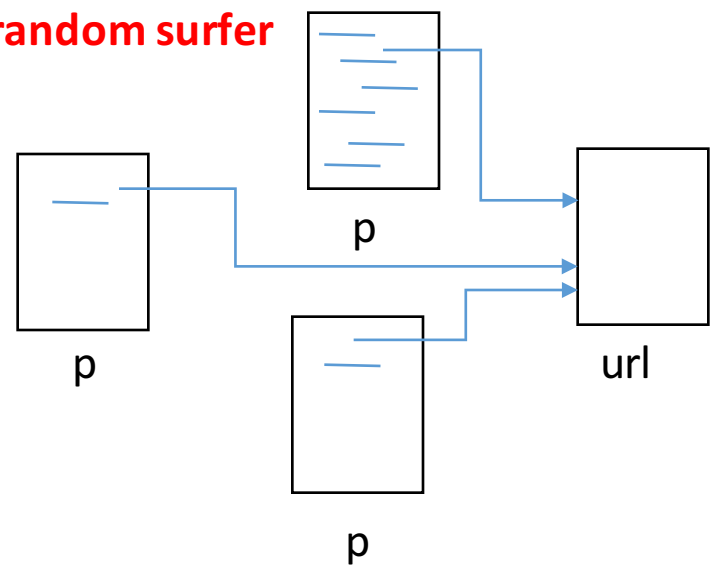
Computing Local Rank

timestep page

$$rank(0, url) \rightarrow 1 \;/\; \mathbf{N}$$

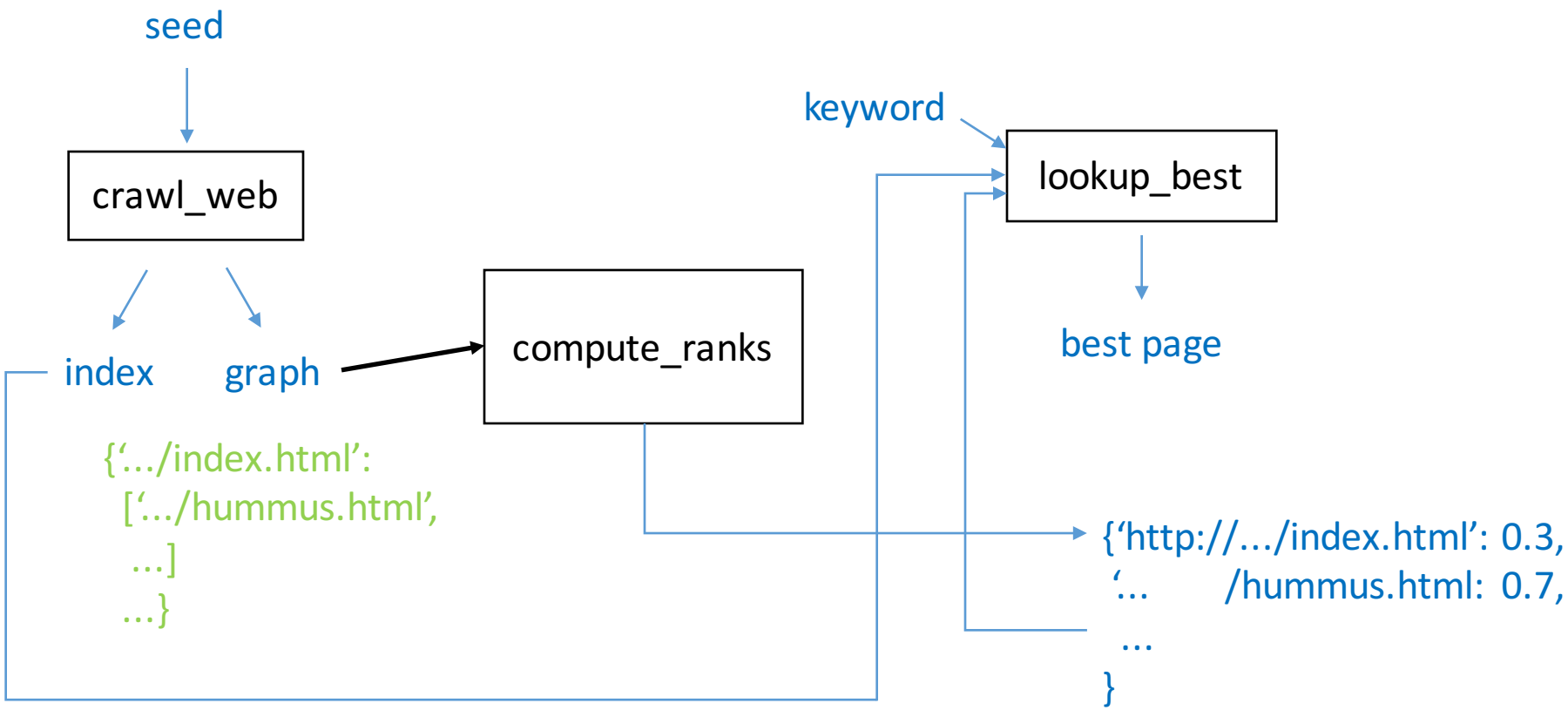
$$rank(t, url) \rightarrow d \sum_{p \in inlinks[url]} (rank(t-1, p) / outlinks[p]) + (1 - d) / N$$

random surfer



d = damping constant
0.8
N = number of pages

Computing Local Rank



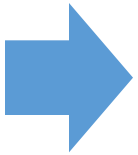
Formal Calculations

$$rank(0, url) = 1 / npages$$

$$rank(t, url) = (1 - d) / npages + \sum_{p \in inlinks[url]} d * rank(t-1, p) / \#\text{ of outlinks from } p$$

d = 0.8

damping



$$newranks[url] = (1 - d) / npages + \sum_{p \in inlinks[url]} d * ranks[p] / \#\text{ of outlinks from } p$$

Dictionaries

- ranks — ranks at time t-1
- newranks — ranks at time t

Computer Ranks

```
def compute_ranks(graph):
    d = 0.8 # damping factor
    numloops = 10

    ranks = {}
    npages = len(graph)
    for page in graph:
        ranks[page] = 1.0 / npages

    for i in range(0, numloops):
        newranks = {}
        for page in graph:
            newrank = (1 - d) / npages
            # update by summing in the inlink ranks

            newranks[page] = newrank
        ranks = newranks
    return ranks
```

$$\text{newrank} = (1 - d) / \text{npages} + \sum_{p \in \text{inlinks[url]}} d * \text{ranks[p]} / \#\text{ of outlinks from p}$$

Quiz: Finishing Local Rank

```
def compute_ranks(graph):
    d = 0.8 # damping factor
    numloops = 10

    ranks = {}
    npages = len(graph)
    for page in graph:
        ranks[page] = 1.0 / npages

    for i in range(0, numloops):
        newranks = {}
        for page in graph:
            newrank = (1 - d) / npages

            #Insert Code Here
            for node in graph:
                if page in graph[node]:
                    newrank = newrank + d * (ranks[node] / len(graph[node]))

            newranks[page] = newrank
        ranks = newranks
    return ranks
```

$$\text{newrank} = (1 - d) / \text{npages} + \sum_{p \in \text{inlinks[url]}} d * \text{ranks[p]} / \#\text{ of outlinks from p}$$

result

```
{'http://udacity.com/cs101x/urank/index.html': 0.033333333333333326,
'http://udacity.com/cs101x/urank/zinc.html': 0.038666666666666655,
'http://udacity.com/cs101x/urank/nickel.html': 0.09743999999999997,
'http://udacity.com/cs101x/urank/kathleen.html': 0.11661866666666663,
'http://udacity.com/cs101x/urank/arsenic.html': 0.05413333333333332,
'http://udacity.com/cs101x/urank/hummus.html': 0.038666666666666655}
```

Search Engine