Introduction

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

Administrative Overview

- VERY IMPORTANT: **course web page**.

  http://vlm1.uta.edu/~athitsos/courses/cse2320_spring2014/

  - The course web page will be the primary source of information about the class.
- To find the course web page:
  - Google my name.
  - Go to my web page.
  - Click on the CSE 2320 link.
- If you have any trouble: E-MAIL ME.

Administrative Overview

- VERY IMPORTANT: **Blackboard**.
  - Blackboard will be the platform for submitting ALL assignments.
  - No submissions via e-mail, or via hard copy in class.
  - If Blackboard says the submission is late, then it is late.
  - Occasionally people submit the wrong files. YOU ARE RESPONSIBLE FOR VERIFYING you submitted the right files, and on time.
- Assignment 0 will be posted today, and due Thursday.
  - It simply checks that you know how to use Blackboard.
  - No credit, the goal is to prevent people saying "I did not know how to use Blackboard" for the first real assignment.

Administrative Overview

- VERY IMPORTANT: **syllabus** (see web page)
  - You are RESPONSIBLE for understanding what the syllabus says, especially if you worry about your grade.
  - The syllabus policies will be STRICTLY followed.

Why Algorithms? An Example

- In 1996, we were working on a web search engine.
- Every day, we had a list A of web pages we have already visited.
  - "visiting" a web page means that our program has downloaded that web page and processed it, so that it can show up in search results.
- Every day, we also had a list B of links to web pages that we still had not processed.
- Question: which links in list B are NOT in A?
- Why was this a useful question?

Why Algorithms? An Example

- In 1996, we were working on a web search engine.
- Every day, we had a list A of web pages we have already visited.
  - "visiting" a web page means that our program has downloaded that web page and processed it, so that it can show up in search results.
- Every day, we also had a list B of links to web pages that we still had not processed.
- Question: which links in list B are NOT in A?
- Why was this a useful question?
  - Most links in B had already been seen in A.
  - It was a **huge waste of resources** to revisit those links.

Why Algorithms? An Example

- Recap:
  - A set A of items
  - A set B of items
  - Define setdiff(B, A) to be the set of items in B that are not in A.
- Question: how do we compute setdiff(B, A).
- Any ideas?

setdiff(B, A) – First Version

```
setdiff(B, A):
   result = empty set
   for each item b of B:
      found = false
      for each item a of A:
         if (b == a) then found = true
      if (found == false) add b to result
   return result.
```

- What can we say about how fast this would run?

setdiff(B, A) – First Version

```
setdiff(B, A):
    result = empty set
    for each item b of B:
        for each item a of A:
            if (b == a) then add b to result
    return result.
```

- This needs to compare each item of B with each item of A.
- If we denote the size of B as |B|, and the size of A as |A|, we need |B| * |A| comparisons.

setdiff(B, A) – First Version

```
setdiff(B, A):
    result = empty set
    for each item b of B:
        for each item a of A:
            if (b == a) then add b to result
    return result.
```

- This needs to compare each item of B with each item of A.
- If we denote the size of B as |B|, and the size of A as |A|, we need |B| * |A| comparisons.
- This is our first analysis of **time complexity**.

setdiff(B, A) – First Version - Speed

- We need to perform |B| * |A| comparisons.
- What does this mean in practice?
- Suppose A has 1 billion items.
- Suppose B has 1 million items.
- We need to do 1 quadrilion comparisons.

setdiff(B, A) – First Version - Speed

- We need to perform |B| * |A| comparisons.
- What does this mean in practice?
- Suppose A has 1 billion items.
- Suppose B has 1 million items.
- We need to do 1 quadrilion comparisons.
- On a computer that can do 1 billion comparisons per second, this would take 11.6 days.
  - This is very optimistic, in practice, it would be at least several months.
  - **CAN WE DO BETTER?**

setdiff(B, A) – Second Version

```
setdiff(B, A):
   result = empty set
   sort A and B in alphabetical order
   i = 0; j = 0
   while (i < size(B)) and (j < size(A)):
      if (B[i] < A[j]) then:
         add B[i] to the result
         i = i+1
      else if (B[i] > a[i]) then j = j+1
      else i = i+1; j = j+1
   while i < size(B):
         add B[i] to result
         i = i+1
   return result
```

Application to an Example

- Suppose:
  - B = {January, February, March, April, May, June, July, August, September, October, November, December}
  - A = {May, August, June, July}
- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = August, B[i] = April.
  - B[i] < A[j]
  - we add B[i] to the result
  - i increases by 1.
- result = {April}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = August, B[i] = August.
  - B[i] equals A[j]
  - i and j both increase by 1.
- result = {April}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = July, B[i] = December.
  - B[i] < A[j]
  - we add B[i] to the result
  - i increases by 1.
- result = {April, December}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = July, B[i] = February.
  - B[i] < A[j]
  - we add B[i] to the result
  - i increases by 1.
- result = {August, December, February}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = July, B[i] = January.
  - B[i] < A[j]
  - we add B[i] to the result
  - i increases by 1.
- result = {August, December, February, January}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = July, B[i] = July.
  - B[i] equals A[j]
  - i and j both increase by 1.
- result = {August, December, February, January}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = June, B[i] = June.
  - B[i] equals A[j]
  - i and j both increase by 1.
- result = {August, December, February, January}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = May, B[i] = March.
  - B[i] < A[j]
  - we add B[i] to the result
  - i increases by 1.
- result = {August, December, February, January, March}

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- A[j] = May, B[i] = May.
  - B[i] equals A[j]
  - i and j both increase by 1.
- result = {August, December, February, January, March}
- What happens next?

Application to an Example

- After sorting in alphabetical order:
  - B = {April, August, December, February, January, July, June, March, May, November, October, September}
  - A = {August, July, June, May}
- We have reached the end of A.
- We add to result the remaining items of B.
- result = {August, December, February, January, March, November, October, September}
- We are done!!!

setdiff(B, A) – Second Version

```
setdiff(B, A):
   result = empty set
   sort A and B in alphabetical order
   i = 0; j = 0
   while (i < size(B)) and (j < size(A)):
      if (B[i] < A[j]) then:
         add B[i] to the result
         i = i+1
      else if (B[i] > a[i]) then j = j+1
      else i = i+1; j = j+1
   while i < size(B):
         add B[i] to result
         i = i+1
   return result
```

- What can we say about its speed? What takes time?

setdiff(B, A) – Second Version - Speed

```
setdiff(B, A):
    result = empty set
    sort A and B in alphabetical order
    i = 0; j = 0
    while (i < size(B)) and (j < size(A)):
        if (B[i] < A[j]) then:
            add B[i] to the result
            i = i+1
        else if (B[i] > a[i]) then j = j+1
        else i = i+1; j = j+1
    while i < size(B):
            add B[i] to result
            i = i+1
    return result
```

- we need to: sort A and B, and execute the while loops.

setdiff(B, A) – Second Version - Speed

- We need to:
  - sort A
  - sort B
  - execute the while loops.
- How many calculations it takes to sort A?
  - We will learn in this class that the number of calculations is |A| * log(|A|) * some unspecified constant.
- How many iterations do the while loops take?
  - no more than |A| + |B|.

setdiff(B, A) – Second Version - Speed

- We will skip some details, since this is just an introductory example.
  - By the end of the course, you will be able to fill in those details.
- It turns out that the number of calculations is proportional to $|A|\log(|A|) + |B|\log(|B|)$.
  - Unless stated otherwise, all logarithms in this course will be base 2.

setdiff(B, A) – Second Version - Speed

- It turns out that the number of calculations is proportional to |A|log(|A|) + |B|log(|B|).
- Suppose A has 1 billion items.
  - log(|A|) = about 30.
- We need to do at least 30 billion calculations (unrealistically optimistic).
- On a computer that can do 1 billion calculations per second, this would take 30 seconds.
  - This is very optimistic, but compare to optimistic estimate of 11.6 days for first version of setdiff.
  - in practice, it would be some minutes, possibly hours, but compare to several months or more for first version.

setdiff(B, A) – Third Version

- Use Hash Tables.
- At this point, you are not supposed to know what hash tables are.
- By the end of the course, you should be able to implement and evaluate all three versions.

Programming Skills vs. Algorithmic Skills

- The setdiff example illustrates the difference between programming skills and algorithmic skills.
- Before taking this course, if faced with the setdiff problem, you should ideally be able to:
  - come up with the first version of the algorithm.
  - implement that version.
- After taking this course, you should be able to come up with the second and third versions, and implement them.

Programming Skills vs. Algorithmic Skills

- Many professional programmers do not know much about algorithms.
- However, even such programmers use non-trivial algorithms all the time (e.g., sorting functions or hash tables).
  - They just rely on built-in functions that already implement such algorithms.
- There are a lot of programming tasks that such programmers are not qualified to work on.

Programming Skills vs. Algorithmic Skills

- A large number of real-world problems are simply impossible to solve without solid algorithmic skills.
  - A small selection of examples: computer and cell phone networks, GPS navigation, search engines, web-based financial transactions, file compression, digital cable TV, digital music and video players, speech recognition, automatic translation, computer games, spell-checking, movie special effects, robotics, spam filtering, …
- Good algorithmic skills give you the ability to work on many really interesting software-related tasks.
- Good algorithmic skills give you the ability to do more scientific-oriented computer-related work.

Next Steps in the Course

- Do a few algorithms, as examples.
- Learn basic methods for analyzing algorithmic properties, such as **time complexity**.
- Learn about some basic data structures, such as linked lists, stacks, and queues.
- Explore, learn and analyze several types of algorithms.
  - Emphasis on sorting, tree algorithms, graph algorithms.
  - Why? Should become a lot clearer as the course progresses.

Up Next: Examples of Algorithms

- Union-Find.
- Binary Search.
- Selection Sort.
- What each of these algorithms does is the next topic we will cover.

Connectivity: An Example

- Suppose that we have a large number of computers, with no connectivity.
  – No computer is connected to any other computer.
- We start establishing direct computer-to-computer links.
- We define connectivity(A, B) as follows:
  – If A and B are directly linked, they are connected.
  – If A and B are connected, and B and C are connected, then A and C are connected.
- Connectivity is *transitive*.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
    - How do we tell the computer? What do we need to provide?

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
    - How do we tell the computer? What do we need to provide?
    - Answer: we need to provide two integers, specifying the two computers that are getting linked.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
    - What does it mean that "connectivity changed"?

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
    - What does it mean that "connectivity changed"?
    - It means that there exist at least two computers X and Y that were not connected before the new link was in place, but are connected now.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
    - Can you come up with an example where the new link does not change connectivity?

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
    - Can you come up with an example where the new link does not change connectivity?
    - Suppose we have computers 1, 2, 3, 4. Suppose 1 and 2 are connected, and 2 and 3 are connected. Then, directly linking 1 to 3 does not add connectivity.

The Union-Find Problem

- We want a program that behaves as follows:
  – Each computer is represented as a number.
  – We start our program.
  – Every time we establish a link between two computers, we tell our program about that link.
  – We want the program to tell us if the new link has changed connectivity or not.
  – How do we do that?

A Useful Connectivity Property

- Suppose we have N computers.
- At each point (as we establish links), these N computers will be divided into separate networks.
  - All computers within a network are connected.
  - If computers A and B belong to different networks, they are not connected.
- Each of these networks is called a **connected component.**

Initial Connectivity

- Suppose we have N computers.
- Before we have established any links, how many connected components do we have?

Initial Connectivity

- Suppose we have N computers.
- Before we have established any links, how many connected components do we have?
  - N components: each computer is its own connected component.

Labeling Connected Components

- Suppose we have N computers.
- Suppose we have already established some links, and we have K connected components.
- How can we keep track, for each computer, what connected component it belongs to?

Labeling Connected Components

- Suppose we have N computers.
- Suppose we have already established some links, and we have K connected components.
- How can we keep track, for each computer, what connected component it belongs to?
  – Answer: maintain an array **id** of N integers.
  – **id[p]** will be the ID of the connected component of computer p (where p is an integer).
  – For convenience, we can establish the convention that the ID of a connected component X is just some integer **p** such that computer **p** belongs to X.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
  - How do we do that?

Union-Find: First Solution

- It is rather straightforward to come up with a brute force method:
- Every time we establish a link between **p** and **q**:
  - The new link means **p** and **q** are connected.
  - If they were already connected, we do not need to do anything.
  - How can we check if they were already connected?

Union-Find: First Solution

- It is rather straightforward to come up with a brute force method:
- Every time we establish a link between **p** and **q**:
  - The new link means **p** and **q** are connected.
  - If they were already connected, we do not need to do anything.
  - How can we check if they were already connected?
    - Answer: **id[p] == id[q]**

Union-Find: First Solution

- It is rather straightforward to come up with a brute force method:
- Every time we establish a link between **p** and **q**:
  - The new link means **p** and **q** are connected.
  - If they were not already connected, then the connected components of **p** and **q** need to be merged.

Union-Find: First Solution

- It is rather straightforward to come up with a brute force method:
- Every time we establish a link between **p** and **q**:
  - The new link means **p** and **q** are connected.
  - If they were not already connected, then the connected components of **p** and **q** need to be merged.
  - We can go through each computer **i** in the network, and if **id[i] == id[p]**, we set **id[i] = id[q]**.

Union-Find: First Solution

```
#include <stdio.h>
#define N 10000
main()
  { int i, p, q, t, id[N];
    for (i = 0; i < N; i++) id[i] = i;
    while (scanf("%d %d\n", &p, &q) == 2)
      {
        if (id[p] == id[q]) continue;
        for (t = id[p], i = 0; i < N; i++)
          if (id[i] == t) id[i] = id[q];
        printf(" %d %d\n", p, q);
      }
 }
```