6.006 Introduction to Algorithms
Spring 2008

Lecture 1: Introduction and the Document Distance Problem

Course Overview

- Efficient procedures for solving problems on large inputs (Ex: entire works of Shakespeare, human genome, U.S. Highway map)
- Scalability
- Classic data structures and elementary algorithms (CLRS text)
- Real implementations in Python $\Leftrightarrow$ Fun problem sets!
- β version of the class - feedback is welcome!

Pre-requisites

- Familiarity with Python and Discrete Mathematics

Contents

The course is divided into 7 modules - each of which has a motivating problem and problem set (except for the last module). Modules and motivating problems are as described below:

1. Linked Data Structures: Document Distance (DD)
2. Hashing: DD, Genome Comparison
3. Sorting: Gas Simulation
4. Search: Rubik's Cube $2 \times 2 \times 2$
5. Shortest Paths: Caltech $\rightarrow$ MIT
6. Dynamic Programming: Stock Market
7. Numerics:$\sqrt{2}$

Document Distance Problem

Motivation

Given two documents, how similar are they?

- Identical - easy?
- Modified or related (Ex: DNA, Plagiarism, Authorship)

- Did Francis Bacon write Shakespeare's plays?

To answer the above, we need to define practical metrics. Metrics are defined in terms of word frequencies.

Definitions

1. *Word*: Sequence of alphanumeric characters. For example, the phrase "6.006 is fun" has 4 words.

2. *Word Frequencies*: Word frequency $D(w)$ of a given word w is the number of times it occurs in a document D.

   For example, the words and word frequencies for the above phrase are as below:

   $$\begin{array}{lcccccc} Count: & 1 & 0 & 1 & 1 & 0 & 1 \\ Word: & 6 & the & is & 006 & easy & fun \end{array}$$

   In practice, while counting, it is easy to choose some canonical ordering of words.

3. *Distance Metric*: The document distance metric is the inner product of the vectors $\boldsymbol{D_1}$ and $\boldsymbol{D_2}$ containing the word frequencies for all words in the 2 documents. Equivalently, this is the projection of vectors $\boldsymbol{D_1}$ onto $\boldsymbol{D_2}$ or vice versa. Mathematically this is expressed as:

   $$\boldsymbol{D_1} \cdot \boldsymbol{D_2} = \sum_w \boldsymbol{D_1}(w) \cdot \boldsymbol{D_2}(w) \tag{1}$$

4. *Angle Metric*: The angle between the vectors $\boldsymbol{D_1}$ and $\boldsymbol{D_2}$ gives an indication of overlap between the 2 documents. Mathematically this angle is expressed as:

   $$\theta(\boldsymbol{D_1}, \boldsymbol{D_2}) = \arccos\left(\frac{\boldsymbol{D_1} \cdot \boldsymbol{D_2}}{\parallel \boldsymbol{D_1} \parallel * \parallel \boldsymbol{D_2} \parallel}\right)$$

   $$0 \leq \theta \leq \pi/2$$

   An angle metric of 0 means the two documents are identical whereas an angle metric of $\pi/2$ implies that there are no common words.

5. *Number of Words in Document*: The magnitude of the vector $\boldsymbol{D}$ which contains word frequencies of all words in the document. Mathematically this is expressed as:

   $$N(\boldsymbol{D}) = \parallel \boldsymbol{D} \parallel = \sqrt{\boldsymbol{D} \cdot \boldsymbol{D}} \tag{2}$$

So let's apply the ideas to a few Python programs and try to flesh out more.

Document Distance in Practice

Computing Document Distance: docdist1.py

The python code and results relevant to this section are available here**.** This program computes the distance between 2 documents by performing the following steps:

- Read file
- Make word list [“the”,“year”,. . .]
- Count frequencies [[“the”,4012],[“year”,55],. . .]
- Sort into order [[“a”,3120],[“after”,17],. . .]
- Compute θ

Ideally, we would like to run this program to compute document distances between writings of the following authors:

- Jules Verne - document size 25k
- Bobsey Twins - document size 268k
- Lewis and Clark - document size 1M
- Shakespeare - document size 5.5M
- Churchill - document size 10M

Experiment: Comparing the Bobsey and Lewis documents with docdist1.py gives $\theta = 0.574$. However, it takes approximately 3 minutes to compute this document distance, and probably gets slower as the inputs get large.
What is wrong with the efficiency of this program?
Is it a Python vs. C issue? Is it a choice of algorithm issue - $\theta(n^2)$ versus $\theta(n)$?

Profiling: docdist2.py

In order to figure out why our initial program is so slow, we now “instrument” the program so that Python will tell us where the running time is going. This can be done simply using the *profile* module in Python. The *profile* module indicates how much time is spent in each routine.
(See this link for details on *profile*).
The *profile* module is imported into docdist1.py and the end of the docdist1.py file is modified. The modified docdist1.py file is renamed as docdist2.py
Detailed results of document comparisons are available here**.**

More on the different columns in the output displayed on that webpage:

- tottime per call(column3) is tottime(column2)/ncalls(column1)
- cumtime(column4)includes subroutine calls
- cumtime per call(column5) is cumtime(column4)/ncalls(column1)

The profiling of the Bobsey vs. Lewis document comparison is as follows:

- Total: 195 secs
- Get words from line list: 107 secs
- Count-frequency: 44 secs
- Get words from string: 13 secs
- Insertion sort: 12 secs

So the get words from line list operation is the culprit. The code for this particular section is:

```
word_list = [ ]
for line in L:
    words_in_line = get_words_from_string(line)
    word_list = word_list + words_in_line
return word_list
```

The bulk of the computation time is to implement

```
word_list = word_list + words_in_line
```

There isn't anything else that takes up much computation time.

List Concatenation: docdist3.py

The problem in docdist1.py as illustrated by docdist2.py is that concatenating two lists takes time proportional to the sum of the lengths of the two lists, since each list is copied into the output list!
$L = L_1 + L_2$ takes time proportional to $\mid L_1 \mid + \mid L_2 \mid$. If we had n lines (each with one word), computation time would be proportional to $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \theta(n^2)$

Solution:

```
word_list.extend (words_in_line)
[word_list.append(word)] for each word in words_in_line
```

Ensures $L_1.extend(L_2)$ time proportional to $\mid L_2 \mid$

Take Home Lesson: Python has powerful primitives (like concatenation of lists) built in. To write efficient algorithms, we need to understand their costs. See Python Cost Model for details. PS1 also has an exercise on figuring out cost of a set of operations.

Incorporate this solution into docdist1.py - rename as docdist3.py. Implementation details and results are available here. This modification helps run the Bobsey vs. Lewis example in 85 secs (as opposed to the original 195 secs).

We can improve further by looking for other quadratic running times hidden in our routines. The next offender (in terms of overall computation time) is the count frequency routine, which computes the frequency of each word, given the word list.

Analysing Count Frequency

```
def count_frequency(word_list):
    """
    Return a list giving pairs of form: (word,frequency)
    """
    L = []
    for new_word in word_list:
        for entry in L:
            if new_word == entry[0]:
                entry[1] = entry[1] + 1
                break
        else:
            L.append([new_word,1])
    return L
```

If document has n words and d distinct words, $\theta(nd)$. If all words distinct, $\theta(n^2)$. This shows that the count frequency routine searches linearly down the list of word/frequency pairs to find the given word. Thus it has quadratic running time! Turns out the count frequency routine takes more than 1/2 of the running time in docdist3.py. Can we improve?

Dictionaries: docdist4.py

The solution to improve the Count Frequency routine lies in hashing, which gives constant running time routines to store and retrieve key/value pairs from a table. In Python, a hash table is called a dictionary. Documentation on dictionaries can be found here.

Hash table is defined a mapping from a domain(finite collection of immutable things) to a range(anything). For example, D[‘ab’] = 2, D[‘the’] = 3.

Modify docdist3.py to docdist4.py using dictionaries to give constant time lookup. Modified count frequency routine is as follows:

```
def count_frequency(word_list):
    """
    Return a list giving pairs of form: (word,frequency)
    """
    D = {}
    for new_word in word_list:
        if D.has_key(new_word):
            D[new_word] = D[new_word]+1
        else:
            D[new_word] = 1
    return D.items()
```

Details of implementation and results are here. Running time is now $\theta(n)$. We have successfully replaced one of our quadratic time routines with a linear-time one, so the running time will scale better for larger inputs. For the Bobsey vs. Lewis example, running time improves from 85 secs to 42 secs.

What's left? The two largest contributors to running time are now:

- Get words from string routine (13 secs) — version 5 of docdist fixes this with translate
- Insertion sort routine (11 secs) — version 6 of docdist fixes this with merge-sort

More on that next time ...

6.006 Introduction to Algorithms
Spring 2008

Lecture 2: More on the Document Distance Problem

Lecture Overview

Today we will continue improving the algorithm for solving the document distance problem.

- Asymptotic Notation: Define notation precisely as we will use it to compare the complexity and efficiency of the various algorithms for approaching a given problem (here Document Distance).
- Document Distance Summary - place everything we did last time in perspective.
- Translate to speed up the 'Get Words from String' routine.
- Merge Sort instead of Insertion Sort routine
  - Divide and Conquer
  - Analysis of Recurrences
- Get rid of sorting altogether?

Readings

CLRS Chapter 4

Asymptotic Notation

General Idea

For any problem (or input), parametrize problem (or input) size as n Now consider many different problems (or inputs) of size n. Then,

$$\begin{aligned} T(n) &= \textit{worst case} \text{ running time for input size n} \\ &= \max_{X\text{: Input of Size n}} \text{running time on } X \end{aligned}$$

How to make this more precise?

- Don't care about $T(n)$ for small n
- Don't care about constant factors (these may come about differently with different computers, languages, ...)

For example, the time (or the number of steps) it takes to complete a problem of size n might be found to be $T(n) = 4n^2 - 2n + 2$ μs. From an asymptotic standpoint, since n^2 will dominate over the other terms as n grows large, we only care about the highest order term. We ignore the constant coefficient preceding this highest order term as well because we are interested in rate of growth.

Formal Definitions

1. **Upper Bound**: We say $T(n)$ is $O(g(n))$ if $\exists\ n_0, \exists\ c$ s.t. $0 \leq T(n) \leq c.g(n)\ \forall n \geq n_0$
   Substituting 1 for n_0, we have $0 \leq 4n^2 - 2n + 2 \leq 26n^2\ \forall n \geq 1$
   $\therefore 4n^2 - 2n + 2 = O(n^2)$
   Some semantics:
   - Read the 'equal to' sign as "is" or ϵ belongs to a set.
   - Read the O as 'upper bound'
2. **Lower Bound**: We say $T(n)$ is $\Omega(g(n))$ if $\exists\ n_0, \exists\ d$ s.t. $0 \leq d.g(n) \leq T(n)\ \forall n \geq n_0$
   Substituting 1 for n_0, we have $0 \leq 4n^2 + 22n - 12 \leq n^2\ \forall n \geq 1$
   $\therefore 4n^2 + 22n - 12 = \Omega(n^2)$
   Semantics:
   - Read the 'equal to' sign as "is" or ϵ belongs to a set.
   - Read the Ω as 'lower bound'
3. **Order**: We say $T(n)$ is $\Theta(g(n))$ iff $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$
   Semantics: Read the Θ as 'high order term is $g(n)$'

Document Distance so far: Review

To compute the 'distance' between 2 documents, perform the following operations:

For each of the 2 files:		
Read file		
Make word list	+ op on list	$\Theta(n^2)$
Count frequencies	double loop	$\Theta(n^2)$
Sort in order	insertion sort, double loop	$\Theta(n^2)$
Once vectors $\boldsymbol{D_1}$,$\boldsymbol{D_2}$ are obtained:		
Compute the angle	$\arccos\left(\frac{\boldsymbol{D_1}\cdot\boldsymbol{D_2}}{\|\boldsymbol{D_1}\|*\|\boldsymbol{D_2}\|}\right)$	$\Theta(n)$

The following table summarizes the efficiency of our various optimizations for the Bobsey vs. Lewis comparison problem:

Version	Optimizations	Time	Asymptotic
V1	initial	?	?
V2	add profiling	195 s	
V3	wordlist.extend(...)	84 s	$\Theta(n^2) \rightarrow \Theta(n)$
V4	dictionaries in count-frequency	41 s	$\Theta(n^2) \rightarrow \Theta(n)$
V5	process words rather than chars in get words from string	13 s	$\Theta(n) \rightarrow \Theta(n)$
V6	merge sort rather than insertion sort	6 s	$\Theta(n^2) \rightarrow \Theta(n \lg(n))$
V6B	eliminate sorting altogether	1 s	a $\Theta(n)$ algorithm

The details for the version 5 (V5) optimization will not be covered in detail in this lecture. The code, results and implementation details can be accessed at this link. The only big obstacle that remains is to replace Insertion Sort with something faster because it takes time $\Theta(n^2)$ in the worst case. This will be accomplished with the Merge Sort improvement which is discussed below.

Merge Sort

Merge Sort uses a divide/conquer/combine paradigm to scale down the complexity and scale up the efficiency of the Insertion Sort routine.

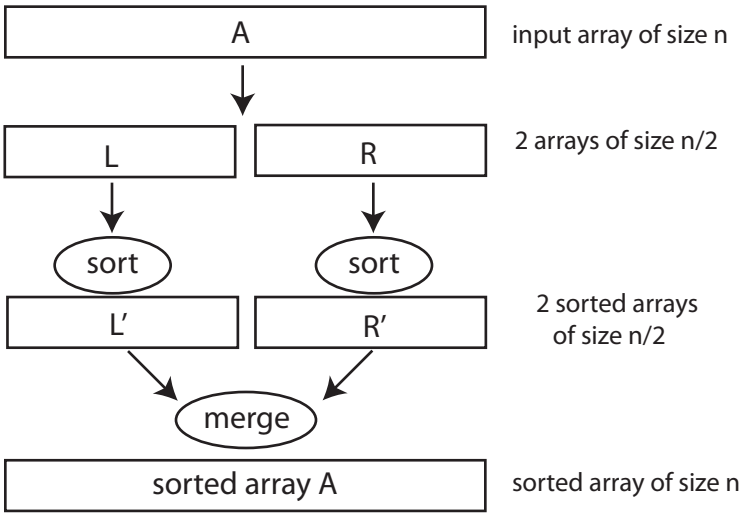


Figure 1: Divide/Conquer/Combine Paradigm

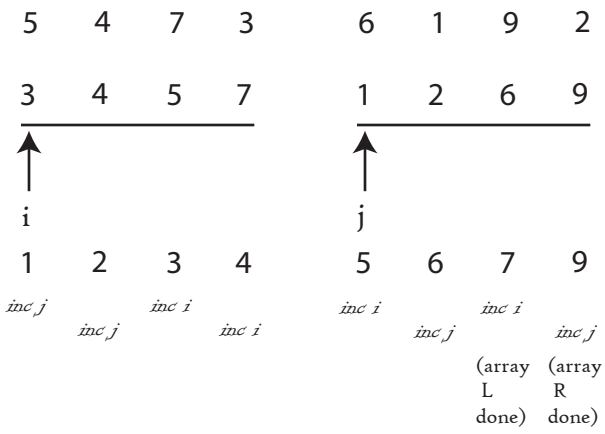


Figure 2: "Two Finger" Algorithm for Merge

The above operations give us $T(n) = \underbrace{C_1}_{divide} + \underbrace{2.T(n/2)}_{recursion} + \underbrace{C.n}_{merge}$

Keeping only the higher order terms,

$$\begin{aligned} T(n) &= 2T(n/2) + C \cdot n \\ &= C \cdot n + 2 \times (C \cdot n/2 + 2(C \cdot (n/4) + \ldots)) \end{aligned}$$

Detailed notes on implementation of Merge Sort and results obtained with this improvement are available here. With Merge Sort, the running time scales "nearly linearly" with the size of the input(s) as $n \lg(n)$ is "nearly linear"in n.

An Experiment

Insertion Sort	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg(n))$ if $n = 2^i$
Built in Sort	$\Theta(n \lg(n))$

- Test Merge Routine: Merge Sort (in Python) takes $\approx 2.2n \lg(n)$ μs
- Test Insert Routine: Insertion Sort (in Python) takes $\approx 0.2n^2$ μs
- Built in Sort or sorted (in C) takes $\approx 0.1n \lg(n)$ μs

The 20X constant factor difference comes about because Built in Sort is written in C while Merge Sort is written in Python.

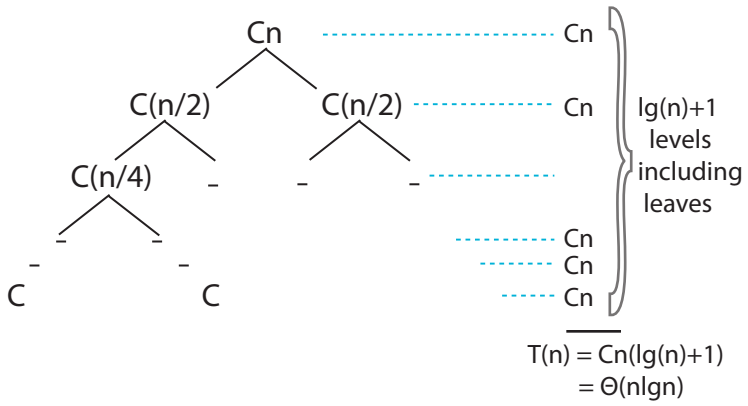


Figure 3: Efficiency of Running Time for Problem of size n is of order $\Theta(n \lg(n))$

Question: When is Merge Sort (in Python) $2n \lg(n)$ better than Insertion Sort (in C) $0.01n^2$?
Aside: Note the 20X constant factor difference between Insertion Sort written in Python and that written in C
Answer: Merge Sort wins for $n \geq 2^{12} = 4096$
Take Home Point: A better algorithm is much more valuable than hardware or compiler even for modest n

See recitation for more Python Cost Model experiments of this sort ...

6.006 Introduction to Algorithms
Spring 2008

Lecture 3: Scheduling and Binary Search Trees

Lecture Overview

- Runway reservation system
  - Definition
  - How to solve with lists
- Binary Search Trees
  - Operations

Readings

CLRS Chapter 10, 12. 1-3

Runway Reservation System

- Airport with single (very busy) runway (Boston $6 \rightarrow 1$)
- "Reservations" for future landings
- When plane lands, it is removed from set of pending events
- Reserve req specify "requested landing time" t
- Add t to the set of no other landings are scheduled within < 3 minutes either way.
  - else error, don't schedule

Example

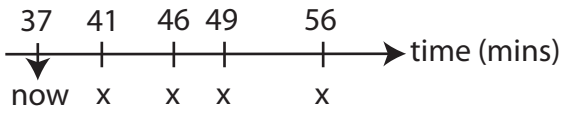


Figure 1: Runway Reservation System Example

Let R denote the reserved landing times: $R = (41, 46, 49, 56)$

Request for time: 44 not allowed ($46 \epsilon R$)
53 OK
20 not allowed (already past)
$| R | = n$

Goal: Run this system efficiently in $O(\lg n)$ time

Algorithm

Keep R as a sorted list.

```
init: R = [ ]
req(t): if t < now: return "error"
for i in range (len(R)):
    if abs(t-R[i]) <3: return "error" %\Theta (n)
R.append(t)
R = sorted(R)
land: t = R[0]
if (t != now) return error
R = R[1: ] (drop R[0] from R)
```

Can we do better?

- **Sorted list**: A 3 minute check can be done in $O(1)$. It is possible to insert new time/plane rather than append and sort but insertion takes $\Theta(n)$ time.
- **Sorted array**: It is possible to do binary search to find place to insert in $O(\lg n)$ time. Actual insertion however requires shifting elements which requires $\Theta(n)$ time.
- **Unsorted list/array**: Search takes $O(n)$ time
- **Dictionary or Python Set**: Insertion is $O(1)$ time. 3 minute check takes $\Omega(n)$ time

What if times are in whole minutes?

Large array indexed by time does the trick. This will not work for arbitrary precision time or verifying width slots for landing.

Key Lesson: Need fast insertion into sorted list.

New Requirement

Rank(t): How many planes are scheduled to land at times $\leq t$? The new requirement necessitates a design amendment.

Binary Search Trees (BST)

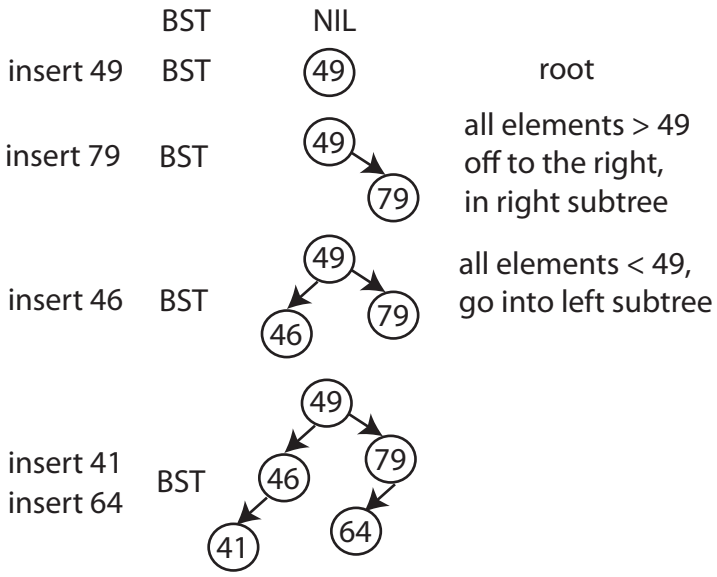


Figure 2: Binary Search Tree

Finding the minimum element in a BST

Key is to just go left till you cannot go left anymore.

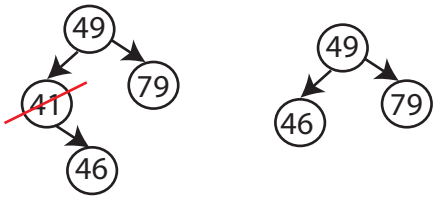


Figure 3: Delete-Min: finds minimum and eliminates it

All operations are $O(h)$ where h is height of the BST.

Finding the next larger element

next-larger(x)

```
if right child not NIL, return minimum(right)
else y = parent(x)
while y not NIL and x = right(y)
    x = y; y = parent(y)
return(y);
```

See Fig. 4 for an example. What would next-larger(46) return?

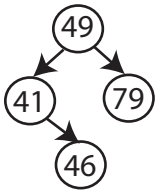


Figure 4: next-larger(x)

What about rank(t)?

Cannot solve it efficiently with what we have but can augment the BST structure.

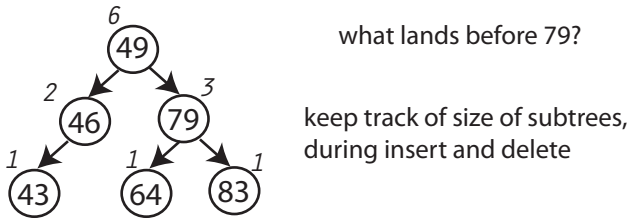


Figure 5: Augmenting the BST Structure

Summarizing from Fig. 5, the algorithm for augmentation is as follows:

1. Walk down tree to find desired time
2. Add in nodes that are smaller
3. Add in subtree sizes to the left

In total, this takes $O(h)$ time.

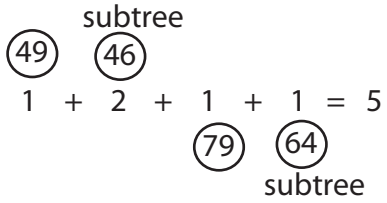


Figure 6: Augmentation Algorithm Example

All the Python code for the Binary Search Trees discussed here are available at this link

Have we accomplished anything?

Height h of the tree should be $O(log(n)$.

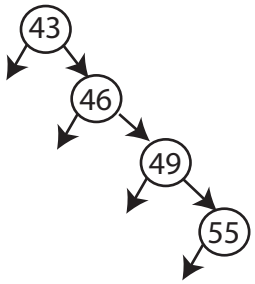


Figure 7: Insert into BST in sorted order

The tree in Fig. 7 looks like a linked list. We have achieved $O(n)$ not $O(log(n)$!!

Balanced BSTs to the rescue...more on that in the next lecture!

6.006 Introduction to Algorithms
Spring 2008

Lecture 4: Balanced Binary Search Trees

Lecture Overview

- The importance of being balanced
- AVL trees
  - Definition
  - Balance
  - Insert
- Other balanced trees
- Data structures in general

Readings

CLRS Chapter 13. 1 and 13. 2 (but different approach: red-black trees)

Recall: Binary Search Trees (BSTs)

- rooted binary tree
- each node has
  - key
  - left pointer
  - right pointer
  - parent pointer

See Fig. 1

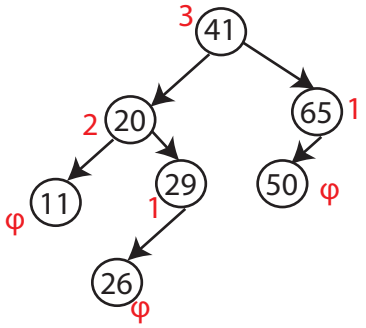


Figure 1: Heights of nodes in a BST

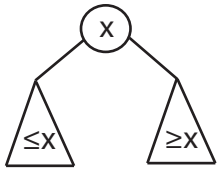


Figure 2: BST property

- BST property (see Fig. 2).
- height of node = length (♯ edges) of longest downward path to a leaf (see CLRS B.5 for details).

The Importance of Being Balanced:

- BSTs support insert, min, delete, rank, etc. in $O(h)$ time, where $h =$ height of tree (= height of root).
- h is between $\lg(n)$ and n: Fig. 3).

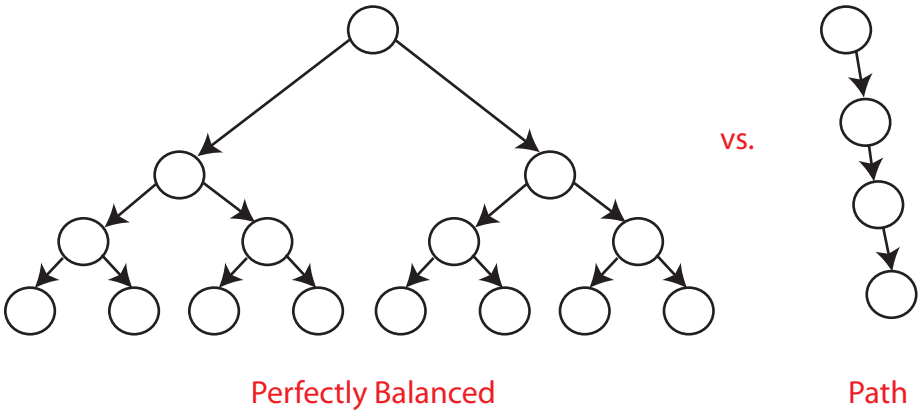


Figure 3: Balancing BSTs

- balanced BST maintains $h = O(\lg n) \Rightarrow$ all operations run in $O(\lg n)$ time.

AVL Trees:

Definition

AVL trees are self-balancing binary search trees. These trees are named after their two inventors G.M. Adel'son-Vel'skii and E.M. Landis [1]
An AVL tree is one that requires heights of left and right children of every node to differ by at most ± 1. This is illustrated in Fig. 4)

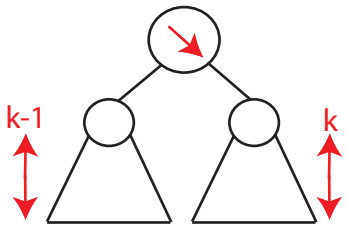


Figure 4: AVL Tree Concept

In order to implement an AVL tree, follow two critical steps:

- Treat nil tree as height -1.
- Each node stores its height. This is inherently a DATA STRUCTURE AUGMENTATION procedure, similar to augmenting subtree size. Alternatively, one can just store difference in heights.

A good animation applet for AVL trees is available at this link. To compare Binary Search Trees and AVL balancing of trees use code provided here.

[1] Original Russian article: Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences 146: 263266. (English translation by Myron J. Ricci in Soviet Math. Doklady, 3:12591263, 1962.)

Balance:

The balance is the worst when every node differs by 1.
Let $N_h =$ min (♯ nodes).

$$\begin{aligned} \Rightarrow N_h &= N_{h-1} + N_{h-2} + 1 \\ &> 2N_{h-2} \\ \Rightarrow N_h &> 2^{h/2} \\ \Longrightarrow h &< \frac{1}{2} \lg h \end{aligned}$$

Alternatively:

$$\begin{aligned} N_h &> F_n && (n^{th} \text{ Fibonacci number}) \\ \text{In fact,} N_h &= F_{n+2} - 1 && (\text{simple induction}) \\ F_h &= \frac{\phi^h}{\sqrt{5}} && (\text{rounded to nearest integer}) \\ \text{where } \phi &= \frac{1+\sqrt{5}}{2} \approx 1.618 && (\text{golden ratio}) \\ \Longrightarrow \max h &\approx \log_\phi(n) \approx 1.440 \lg(n) \end{aligned}$$

AVL Insert:

1. insert as in simple BST
2. work your way up tree, restoring AVL property (and updating heights as you go).

Each Step:

- suppose x is lowest node violating AVL
- assume x is right-heavy (left case symmetric)
- if x's right child is right-heavy or balanced: follow steps in Fig. 5
- else follow steps in Fig. 6
- then continue up to x's grandparent, greatgrandparent ...

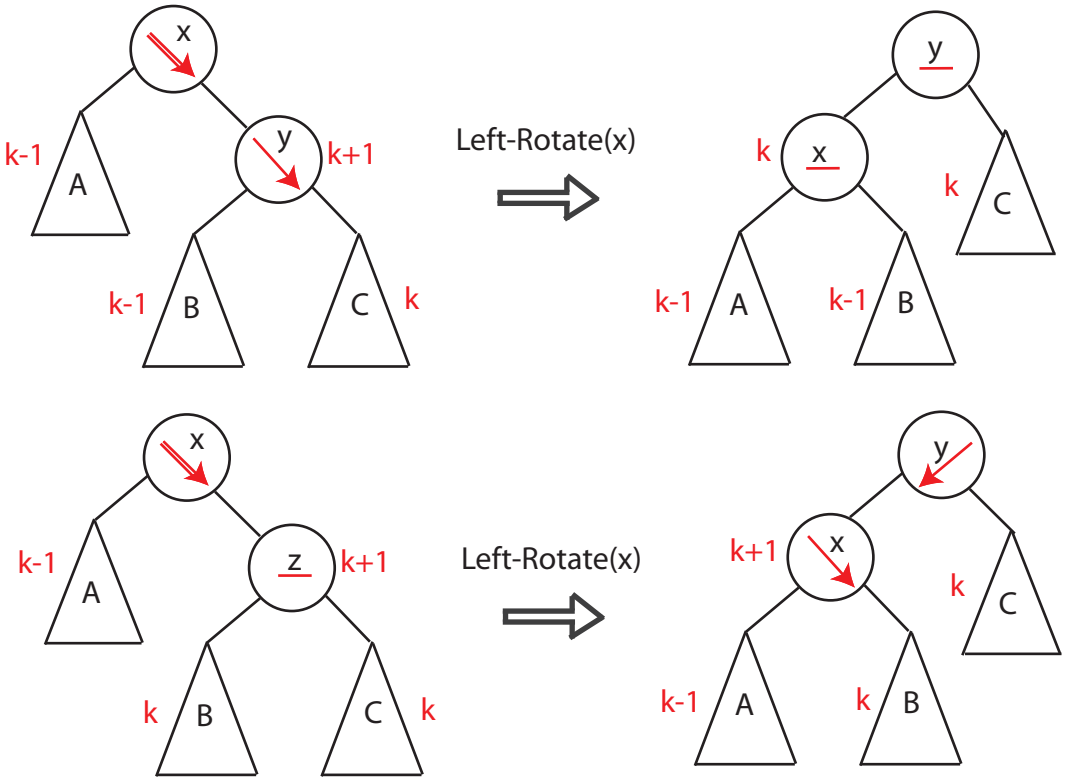


Figure 5: AVL Insert Balancing

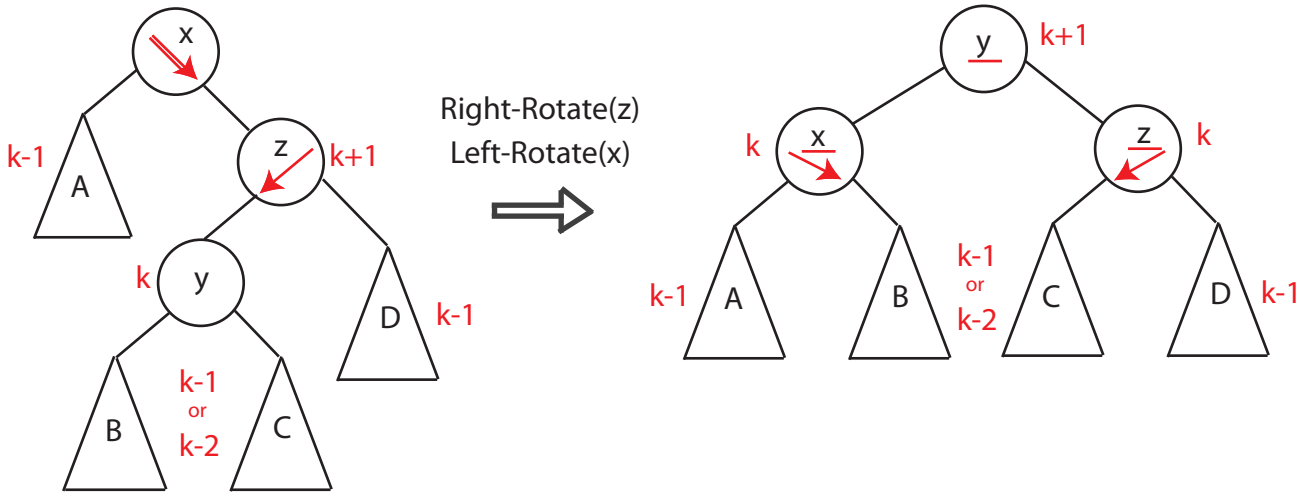


Figure 6: AVL Insert Balancing

Example: An example implementation of the AVL Insert process is illustrated in Fig. 7

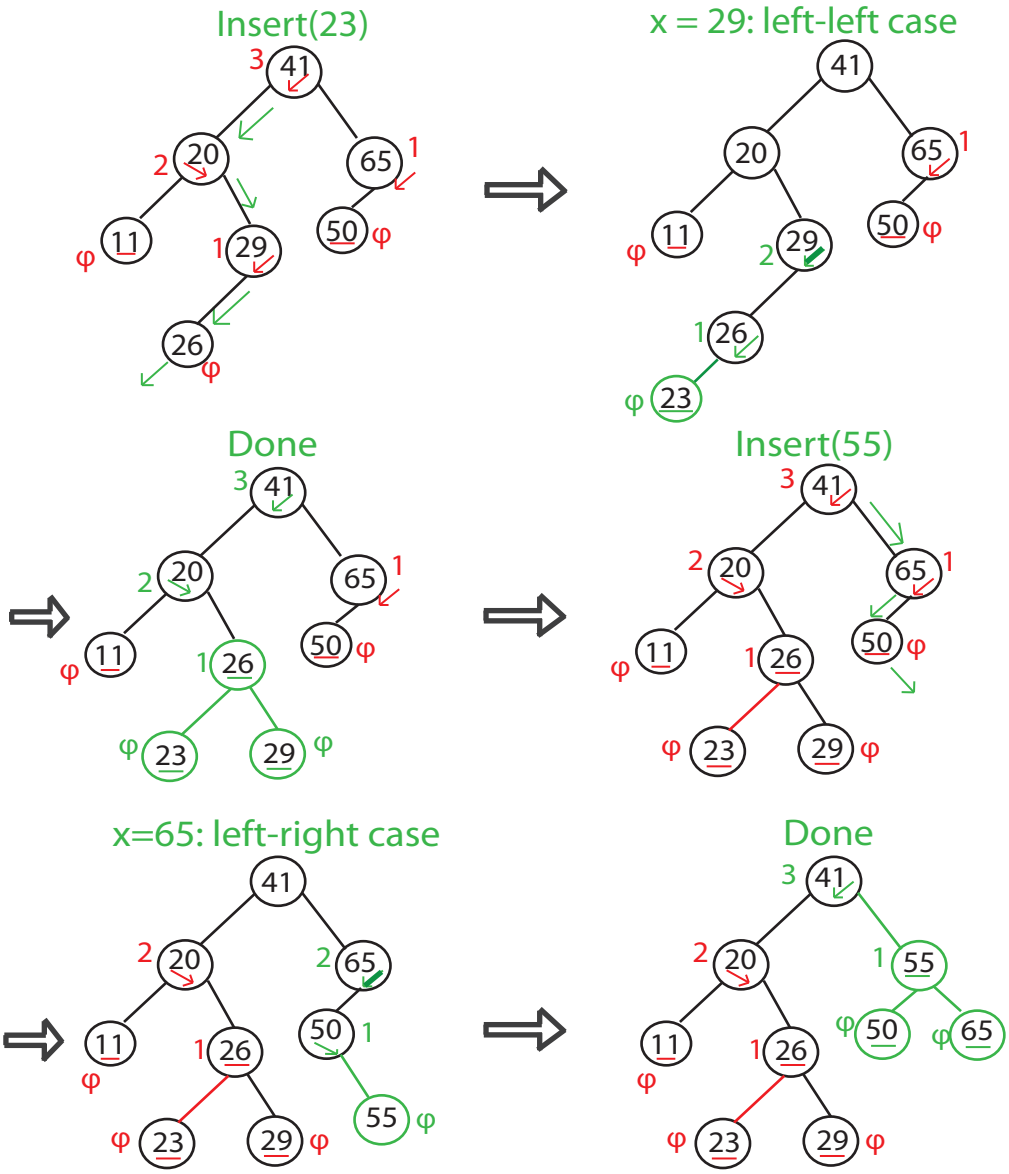


Figure 7: Illustration of AVL Tree Insert Process

Comment 1. In general, process may need several rotations before an Insert is completed.

Comment 2. Delete(-min) harder but possible.

Balanced Search Trees:

There are many balanced search trees.

AVL Trees	Adel'son-Velsii and Landis 1962
B-Trees/2-3-4 Trees	Bayer and McCreight 1972 (see CLRS 18)
BB[α] Trees	Nievergelt and Reingold 1973
Red-black Trees	CLRS Chapter 13
Splay-Trees	Sleator and Tarjan 1985
Skip Lists	Pugh 1989
Scapegoat Trees	Galperin and Rivest 1993
Treaps	Seidel and Aragon 1996

Note 1. Skip Lists and Treaps use random numbers to make decisions fast with high probability.

Note 2. Splay Trees and Scapegoat Trees are "amortized": adding up costs for several operations $\implies$ fast on average.

Splay Trees

Upon access (search or insert), move node to root by sequence of rotations and/or double-rotations (just like AVL trees). Height can be linear but still $O(\lg n)$ per operation "on average" (amortized)

Note: We will see more on amortization in a couple of lectures.

Optimality

- For BSTs, cannot do better than $O(\lg n)$ per search in worst case.
- In some cases, can do better e.g.
  - in-order traversal takes $\Theta(n)$ time for n elements.
  - put more frequent items near root

  A Conjecture: Splay trees are O(best BST) for every access pattern.
- With fancier tricks, can achieve $O(\lg \lg u)$ performance for integers $1 \cdots u$ [Van Ernde Boas; see 6.854 or 6.851 (Advanced Data Structures)]

Big Picture:

Abstract Data Type(ADT): interface spec.
e.g. *Priority Queue*:

- Q = new-empty-queue()
- Q.insert(x)
- x = Q.deletemin()

vs.

Data Structure (DS): algorithm for each op.

There are many possible DSs for one ADT. One example that we will discuss much later in the course is the "heap" priority queue.

6.006 Introduction to Algorithms
Spring 2008

Lecture 5: Hashing I: Chaining, Hash Functions

Lecture Overview

- Dictionaries and Python
- Motivation
- Hash functions
- Chaining
- Simple uniform hashing
- "Good" hash functions

Readings

CLRS Chapter 11. 1, 11. 2, 11. 3.

Dictionary Problem

Abstract Data Type (ADT) maintains a set of items, each with a key, subject to

- insert(item): add item to set
- delete(item): remove item from set
- search(key): return item with key if it exists
- assume items have distinct keys (or that inserting new one clobbers old)
- balanced BSTs solve in $O(\lg n)$ time per op. (in addition to inexact searches like nextlargest).
- goal: $O(1)$ time per operation.

Python Dictionaries:

Items are (key, value) pairs e.g. $d =$ 'algorithms': 5, 'cool': 42

d.items()	→	[('algorithms', 5),('cool',5)]
d['cool']	→	42
d[42]	→	KeyError
'cool' in d	→	True
42 in d	→	False

Python set is really dict where items are keys.

Motivation

Document Distance

- already used in

```
def count_frequency(word_list):
D = {}
for word in word_list:
    if word in D:
        D[word] += 1
    else:
        D[word] = 1
```

- new docdist7 uses dictionaries instead of sorting:

```
def inner_product(D1, D2):
sum = φ. φ

for key in D1:
     if key in D2:
         sum += D1[key]*D2[key]
```

$\Longrightarrow$ optimal $\Theta(n)$ document distance *assuming* dictionary ops. take $O(1)$ time

PS2

How close is chimp DNA to human DNA?
= Longest common substring of two strings
e.g. ALGORITHM vs. ARITHMETIC.

Dictionaries help speed algorithms e.g. put all substrings into set, looking for duplicates - $\Theta(n^2)$ operations.

How do we solve the dictionary problem?

A simple approach would be a direct access table. This means items would need to be stored in an array, indexed by key.

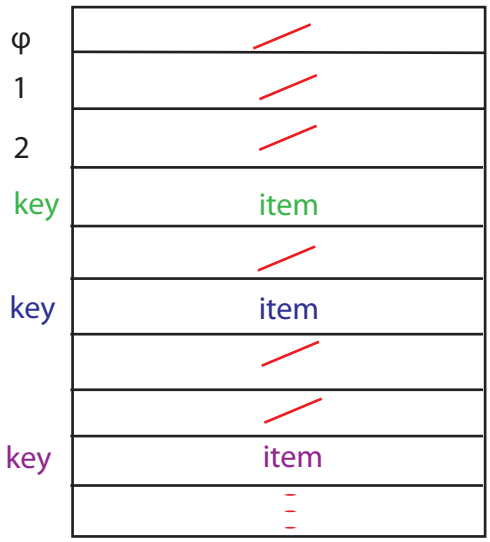


Figure 1: Direct-access table

Problems:

1. keys must be nonnegative integers (or using two arrays, integers)
2. large key range $\implies$ large space e.g. one key of 2^{256} is bad news.

2 Solutions:

Solution 1: map key space to integers.

- In Python: hash (object) where object is a number, string, tuple, etc. or object implementing — hash — Misnomer: should be called "prehash"
- Ideally, $x = y \Leftrightarrow \text{hash}(x) = \text{hash}\ (y)$
- Python applies some heuristics e.g. hash('$\backslash\phi B$ ') = 64 = hash('$\backslash\phi \backslash\ \phi C$')
- Object's key should not change while in table (else cannot find it anymore)
- No mutable objects like lists

Solution 2: hashing (verb from 'hache' = hatchet, Germanic)

- Reduce universe U of all keys (say, integers) down to reasonable size m for table
- idea: $m \approx n$, $n =\mid k \mid$, $k =$ keys in dictionary
- hash function h: $U \rightarrow \phi, 1, \ldots, m-1$

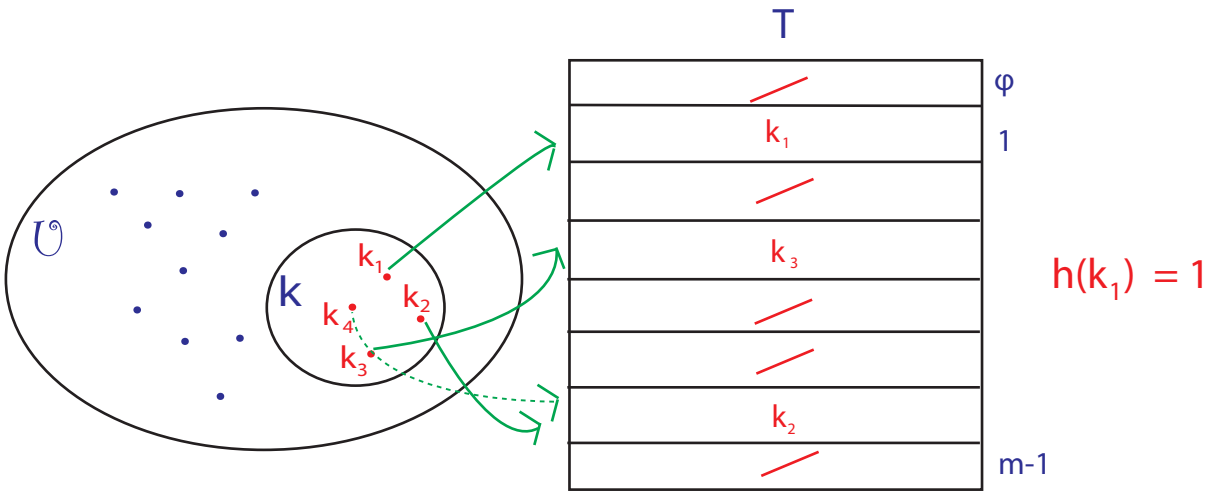


Figure 2: Mapping keys to a table

- two keys $k_i, k_j \epsilon\, K$ collide if $h(k_i) = h(k_j)$

***How* do we deal with collisions?**

There are two ways

1. Chaining: TODAY
2. Open addressing: NEXT LECTURE

Chaining

Linked list of colliding elements in each slot of table

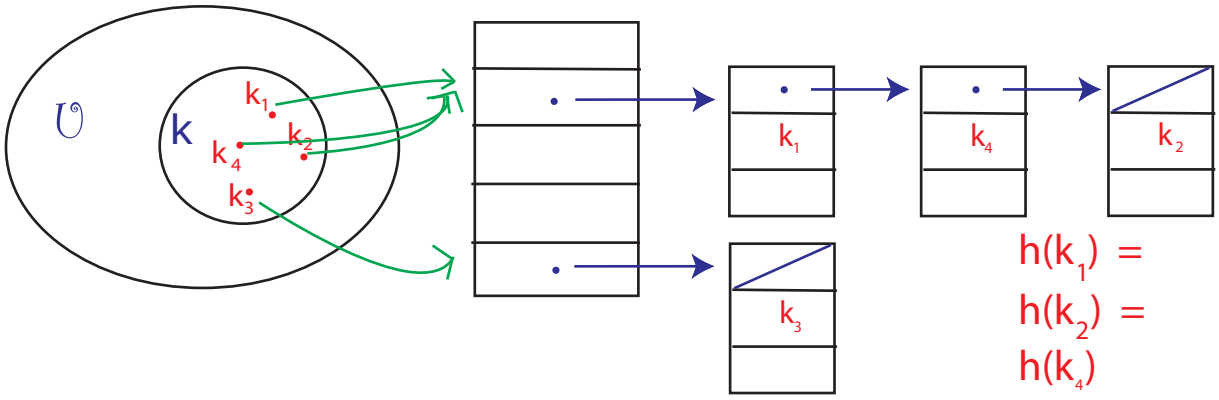


Figure 3: Chaining in a Hash Table

- Search must go through *whole* list T[h(key)]
- Worst case: all keys in k hash to same slot $\implies \Theta(n)$ per operation

Simple Uniform Hashing - an Assumption:

Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed.

$$\begin{aligned} \text{let } n &= \sharp \text{keys stored in table} \\ m &= \sharp \text{slots in table} \\ \underline{\text{load factor}}\ \alpha &= n/m = \text{average } \sharp \text{ keys per slot} \end{aligned}$$

Expected performance of chaining: assuming simple uniform hashing

The performance is likely to be $O(1+\alpha)$ - the 1 comes from applying the hash function and access slot whereas the α comes from searching the list. It is actually $\Theta(1+\alpha)$, even for successful search (*see CLRS*).

Therefore, the performance is $O(1)$ if $\alpha = O(1)$ i. e. $m = \Omega(n)$.

Hash Functions

Division Method:

$h(k) = k \bmod m$

- k_1 and k_2 collide when $k_1 = k_2 (\bmod m)$ i. e. when m divides $| k_1 - k_2 |$
- fine if keys you store are uniform random
- but if keys are $x, 2x, 3x, \ldots$ (regularity) and x and m have common divisor d then use only $1/d$ of table. This is likely if m has a small divisor e. g. 2.
- if $m = 2^r$ then only look at r bits of key!

Good Practice: A good practice to avoid common regularities in keys is to make m a prime number that is not close to power of 2 or 10.
Key Lesson: It is inconvenient to find a prime number; division slow.

Multiplication Method:

$h(k) = [(a \cdot k) \bmod 2^w] \gg (w - r)$ where $m = 2^r$ and w-bit machine words and $a =$ odd integer between $2^(w-1)$ and 2^w.
Good Practise: a not too close to $2^{(w-1)}$ or 2^w.
Key Lesson: Multiplication and bit extraction are faster than division.

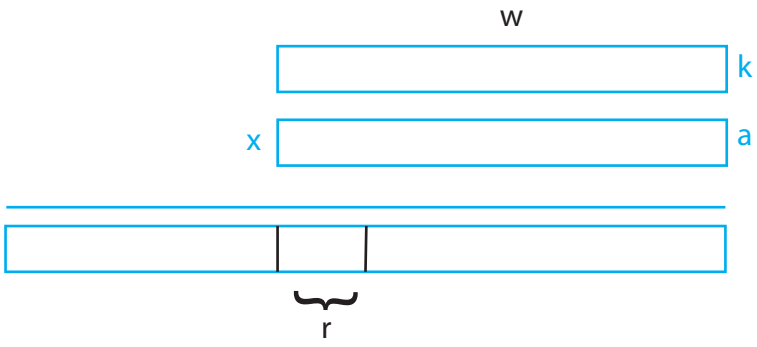


Figure 4: Multiplication Method

6.006 Introduction to Algorithms
Spring 2008

Lecture 6: Hashing II: Table Doubling, Karp-Rabin

Lecture Overview

- Table Resizing
- Amortization
- String Matching and Karp-Rabin
- Rolling Hash

Readings

CLRS Chapter 17 and 32.2.

Recall:

Hashing with Chaining:

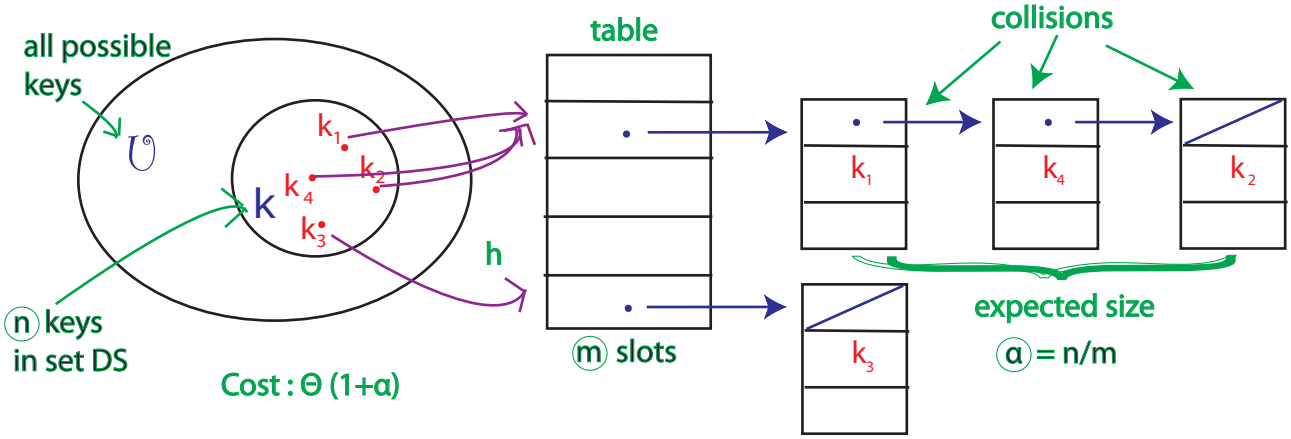


Figure 1: Chaining in a Hash Table

Multiplication Method:

$$\begin{aligned} h(k) &= [(a \cdot k) \bmod 2^w] \gg (w - r) \\ \text{where } m &= \text{table size } = 2^r \\ w &= \text{number of bits in machine words} \\ a &= \text{odd integer between } 2^{w-1} \text{ and } 2^w \end{aligned}$$

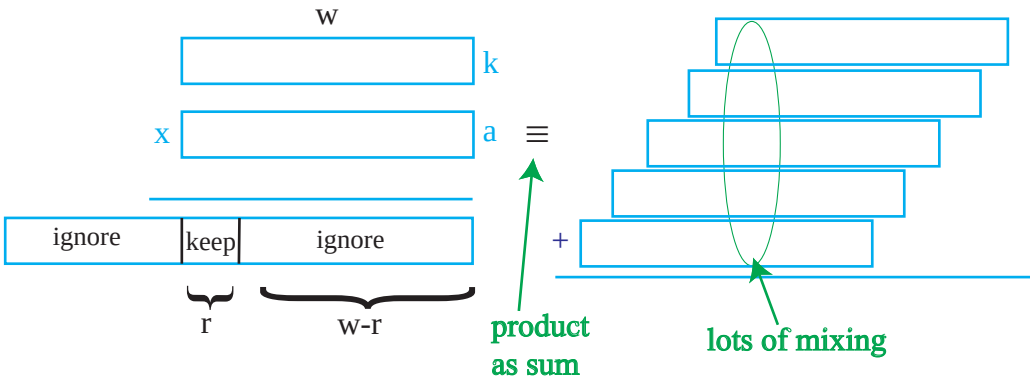


Figure 2: Multiplication Method

How Large should Table be?

- want $m = \theta(n)$ at all times
- don't know how large n will get at creation
- m too small $\implies$ slow; m too big $\implies$ wasteful

Idea:

Start small (constant) and grow (or shrink) as necessary.

Rehashing:

To grow or shrink table hash function must change (m, r)

$\implies$ must rebuild hash table from scratch
for item in old table:
insert into new table
$\implies$ $\Theta(n + m)$ time $= \Theta(n)$ if $m = \Theta(n)$

How fast to grow?

When n reaches m, say

- $m+=1$?
  $\implies$ rebuild every step
  $\implies$ n inserts cost $\Theta(1+2+\cdots+n) = \Theta(n^2)$
- $m*=2$? $m = \Theta(n)$ still $(r+=1)$
  $\implies$ rebuild at insertion 2^i
  $\implies$ n inserts cost $\Theta(1+2+4+8+\cdots+n)$ where n is really the next power of 2
  $= \Theta(n)$
- a few inserts cost linear time, but $\Theta(1)$ "on average".

Amortized Analysis

This is a common technique in data structures - like paying rent: \$ 1500/month $\approx$ \$ 50/day

- operation has amortized cost $T(n)$ if k operations cost $\leq k \cdot T(n)$
- "$T(n)$ amortized" roughly means $T(n)$ "on average", but averaged over all ops.
- e.g. inserting into a hash table takes $O(1)$ amortized time.

Back to Hashing:

Maintain $m = \Theta(n)$ so also support search in $O(1)$ expected time assuming simple uniform hashing

Delete:

Also $O(1)$ expected time

- space can get big with respect to n e.g. $n\times$ insert, $n\times$ delete
- solution: when n decreases to $m/4$, shrink to half the size $\implies$ $O(1)$ amortized cost for both insert and delete - analysis is harder; (see CLRS 17.4).

String Matching

Given two strings s and t, does s occur as a substring of t? (and if so, where and how many times?)
E.g. $s =$ '6.006' and $t =$ your entire INBOX ('grep' on UNIX)

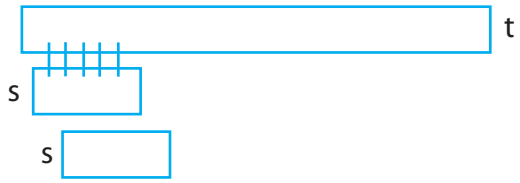


Figure 3: Illustration of Simple Algorithm for the String Matching Problem

Simple Algorithm:

Any ($s == t[i : i + \text{len(s)}]$ for i in range(len(t)-len(s)))
- $O(|\ s\ |)$ time for each substring comparison
$\implies O(|\ s\ | \cdot (|\ t\ | - |\ s\ |))$ time
$= O(|\ s\ | \cdot |\ t\ |)$ potentially quadratic

Karp-Rabin Algorithm:

- Compare $h(s) == h(t[i : i + \text{len(s)}])$
- If hash values match, likely so do strings
  - can check $s == t[i : i + \text{len(s)}]$ to be sure $\sim$ cost $O(|\ s\ |)$
  - if yes, found match — done
  - if no, happened with probability $< \frac{1}{|s|}$
    $\implies$ expected cost is $O(1)$ per i.
- need suitable hash function.
- expected time $= O(|\ s\ | + |\ t\ | \cdot cost(h))$.
  - naively $h(x)$ costs $|\ x\ |$
  - we'll achieve $O(1)$!
  - idea: $t[i : i + len(s)] \approx t[i + 1 : i + 1 + len(s)]$.

Rolling Hash ADT

Maintain string subject to

- h(): reasonable hash function on string
- h.append(c): add letter c to end of string
- h.skip(c): remove front letter from string, assuming it is c

Karp-Rabin Application:

```
for c in s: hs.append(c)
for c in t[:len(s)]:ht.append(c)
if hs() == ht(): ...
```

This first block of code is $O(|\ s\ |)$

```
for i in range(len(s), len(t)):
   ht.skip(t[i-len(s)])
   ht.append(t[i])
   if hs() == ht(): ...
```

The second block of code is $O(|\ t\ |)$

Data Structure:

Treat string as a multidigit number u in base a where a denotes the alphabet size. E.g. 256

- $h() = u \bmod p$ for prime $p \approx |\ s\ |$ or $|\ t\ |$ (division method)
- h stores $u \bmod p$ and $|\ u\ |$, not u

  $\Longrightarrow$ smaller and faster to work with ($u \bmod p$ fits in one machine word)
- h.append(c): $(u \cdot a + \text{ ord }(c)) \bmod p = [(u \bmod p) \cdot a + \text{ ord }(c)] \bmod p$
- h.skip(c): $[u - \text{ ord }(c) \cdot (a^{|u|-1} \bmod p)] \bmod p$

  $= [(u \bmod p) - \text{ ord }(c) \cdot (a^{|u-1|} \bmod p)] \bmod p$

6.006 Introduction to Algorithms
Spring 2008

Lecture 7: Hashing III: Open Addressing

Lecture Overview

- Open Addressing, Probing Strategies
- Uniform Hashing, Analysis
- Advanced Hashing

Readings

CLRS Chapter 11.4 (and 11.3.3 and 11.5 if interested)

Open Addressing

Another approach to collisions

- no linked lists
- all items stored in table (see Fig. 1)

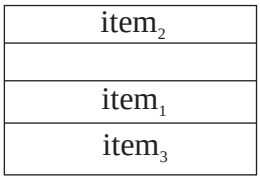


Figure 1: Open Addressing Table

- one item per slot $\implies m \geq n$
- hash function specifies order of slots to probe (try) for a key, not just one slot: (see Fig. 2)

Insert(k,v)

```
for i in xrange(m):
    if T[h(k,i)] is None:        # empty slot
        T[h(k,i)] = (k,v)        # store item
        return
raise 'full'
```

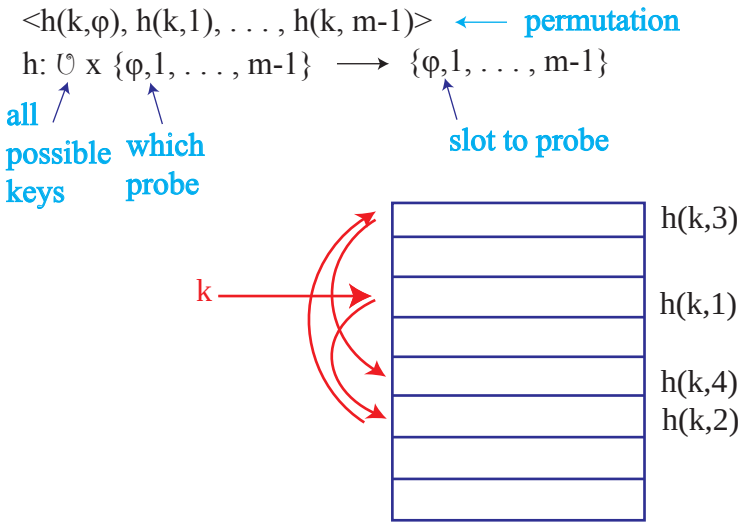


Figure 2: Order of Probes

Example: Insert $k = 496$

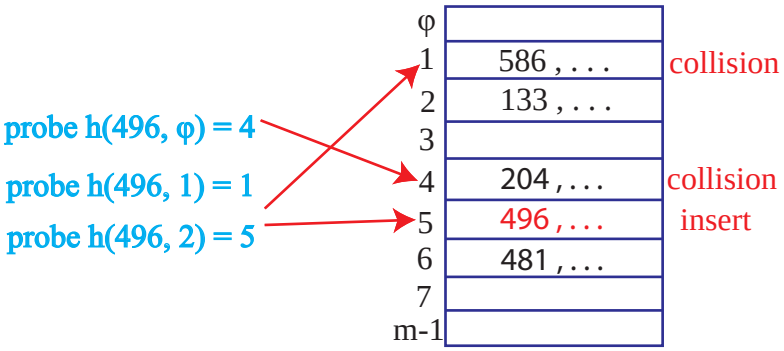


Figure 3: Insert Example

Search(k)

```
for i in xrange(m):
    if   T[h(k, i)] is None:        # empty slot?
         return None                # end of "chain"
    elif T[h(k, i)][φ] == k:        # matching key
         return T[h(k, i)]          # return item
return None                         # exhausted table
```

Delete(k)

- can't just set $T[h(k,i)] = \text{None}$
- *example*: delete(586) $\implies$ search(496) fails
- replace item with DeleteMe, which Insert treats as None but Search doesn't

Probing Strategies

Linear Probing

$h(k,i) = (h'(k) + i) \bmod m$ where $h'(k)$ is ordinary hash function

- like street parking
- problem: *clustering* as consecutive group of filled slots grows, gets *more* likely to grow (see Fig. 4)

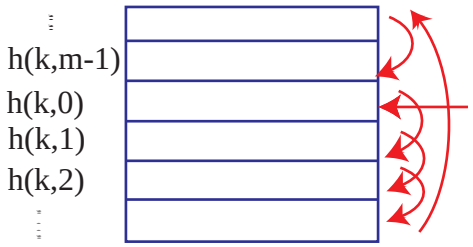


Figure 4: Primary Clustering

- for $0.01 < \alpha < 0.99$ say, clusters of $\Theta(\lg n)$. These clusters are known
- for $\alpha = 1$, clusters of $\Theta(\sqrt{n})$ These clusters are known

Double Hashing

$h(k,i) = (h_1(k) + i.\ h_2(k)) \bmod m$ where $h_1(k)$ and $h_2(k)$ are two ordinary hash functions.

- actually hit all slots (permutation) if $h_2(k)$ is relatively prime to m
- e.g. $m = 2^r$, make $h_2(k)$ always odd

Uniform Hashing Assumption

Each key is equally likely to have any one of the $m!$ permutations as its probe sequence

- not really true
- but double hashing can come close

Analysis

Open addressing for n items in table of size m has expected cost of $\leq \frac{1}{1-\alpha}$ per operation, where $\alpha = n/m (< 1)$ assuming uniform hashing
Example: $\alpha = 90\% \implies$ 10 expected probes

Proof:

Always make a first probe.
With probability n/m, first slot occupied.
In worst case (e.g. key not in table), go to next.
With probability $\frac{n-1}{m-1}$, second slot occupied.
Then, with probability $\frac{n-2}{m-2}$, third slot full.
Etc. (n possibilities)

$$\begin{aligned}
\text{So expected cost} &= 1 + \frac{n}{m}(1 + \frac{n-1}{m-1}(1 + \frac{n-2}{m-2}(\cdots) \\
\text{Now } \frac{n-1}{m-1} \leq \frac{n}{m} &= \alpha \text{ for } i = \phi, \cdots, n (\leq m) \\
\text{So expected cost} &\leq 1 + \alpha(1 + \alpha(1 + \alpha(\cdots))) \\
&= 1 + \alpha + \alpha^2 + \alpha^3 + \cdots \\
&= \frac{1}{1-\alpha}
\end{aligned}$$

Open Addressing vs. Chaining

Open Addressing: better cache performance and rarely allocates memory

Chaining: less sensitive to hash functions and α

Advanced Hashing

Universal Hashing

Instead of defining one hash function, define a whole family and select one at random

- e.g. multiplication method with *random* a
- can prove Pr (over random h) $\{h(x) = h(y)\} = \frac{1}{m}$ for *every (i.e. not random)* $x \neq y$
- $\implies$ $O(1)$ expected time per operation without assuming simple uniform hashing! CLRS 11.3.3

Perfect Hashing

Guarantee $O(1)$ worst-case search

- idea: if $m = n^2$ then E[♯ collisions] $\approx \frac{1}{2}$
  $\implies$ get ϕ after $O(1)$ tries . . . but $O(n^2)$ space
- use this structure for storing chains

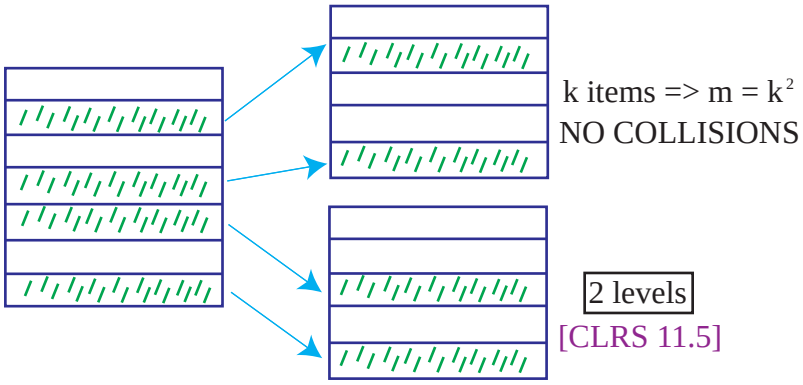


Figure 5: Two-level Hash Table

- can prove $O(n)$ expected total space!
- if ever fails, rebuild from scratch

6.006 Introduction to Algorithms
Spring 2008

Lecture 8: Sorting I: Heaps

Lecture Overview

- Review: Insertion Sort and Merge Sort
- Selection Sort
- Heaps

Readings

CLRS 2.1, 2.2, 2.3, 6.1, 6.2, 6.3 and 6.4

Sorting Review

Insertion Sort

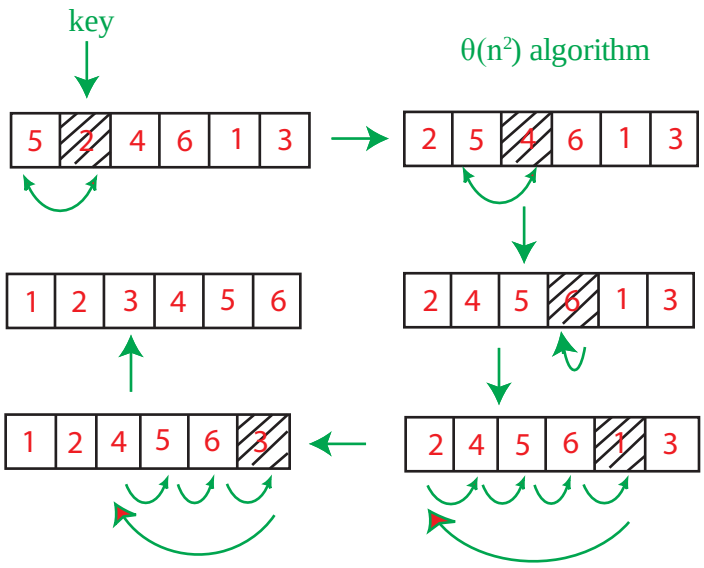


Figure 1: Insertion Sort Example

Merge Sort

Divide n-element array into two subarrays of $n/2$ elements each. Recursively sort sub-arrays using mergesort. Merge two sorted subarrays.

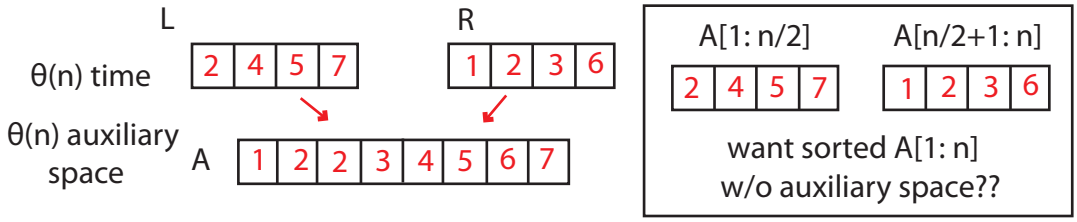


Figure 2: Merge Sort Example

In-Place Sorting

Numbers re-arranged in the array A with at most a *constant* number of them sorted outside the array at any time.

Insertion Sort: stores key outside array $\Theta(n^2)$ in-place

Merge Sort: Need $O(n)$ auxiliary space $\Theta(n \lg n)$ during merging

Question: Can we have $\Theta(n \lg n)$ in-place sorting?

Selection Sort

0. $i = 1$
1. Find minimum value in list beginning with i
2. Swap it with the value in i^{th} position
3. $i = i + 1$, stop if $i = n$

Iterate steps 0-3 n times. Step 1 takes $O(n)$ time. Can we improve to $O(\lg n)$?

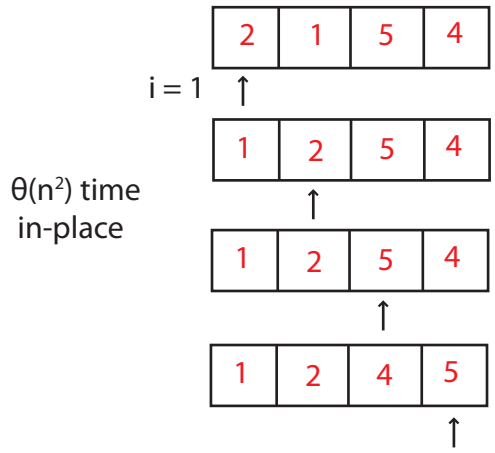


Figure 3: Selection Sort Example

Heaps (Not garbage collected storage)

A heap is an array object that is viewed as a nearly complete binary tree.

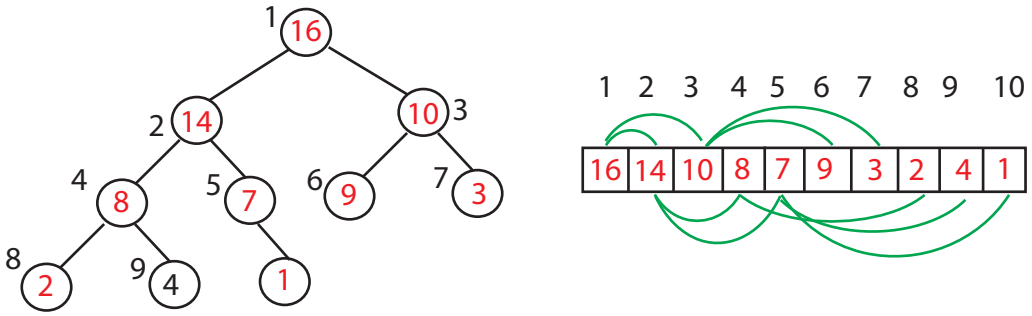


Figure 4: Binary Heap

Data Structure

root $A[i]$
Node with index i
PARENT(i) $= \lfloor \frac{i}{2} \rfloor$
LEFT(i) $= 2i$
RIGHT(i) $= 2i + 1$

Note: NO POINTERS!

length[A]: number of elements in the array

heap-size[A]: number of elements in the heap stored within array A

heap-size[A]: $\leq$ length[A]

Max-Heaps and Min-Heaps

Max-Heap Property: For every node i other than the root A[PARENT(i)] $\geq A[i]$
Height of a binary heap $O(\lg n)$

MAX_HEAPIFY: $O(\lg n)$ maintains max-heap property

BUILD_MAX_HEAP: $O(n)$ produces max-heap from unordered input array

HEAP_SORT: $O(n \lg n)$

Heap operations insert, extract_max etc $O(\lg n)$.

Max_Heapify(A,i)

$l \leftarrow$ left(i)
$r \leftarrow$ right(i)
if $l \leq$ heap-size(A) and $A[l] > A[i]$
  then largest $\leftarrow l$
  else largest $\leftarrow i$
if $r \leq$ heap-size(A) and $A[r] >$ largest
  then largest $\leftarrow r$
if largest $\neq i$
  then exchange $A[i]$ and A[largest]
    MAX_HEAPIFY(A, largest)

This assumes that the trees rooted at left(i) and Right(i) are max-heaps. $A[i]$ may be smaller than children violating max-heap property. Let the $A[i]$ value "float down" so subtree rooted at index i becomes a max-heap.

Example

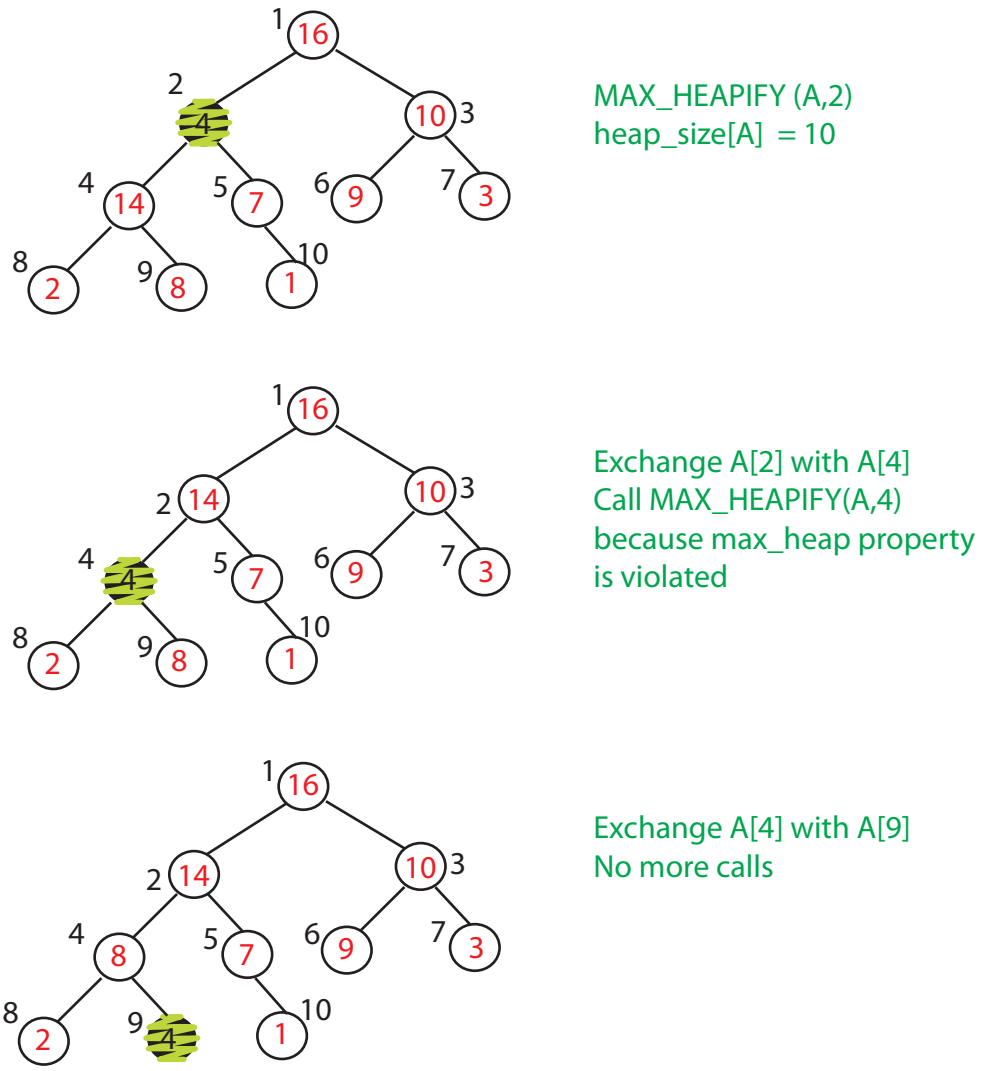


Figure 5: MAX HEAPIFY Example

6.006 Introduction to Algorithms
Spring 2008

Lecture 9: Sorting II: Heaps

Lecture Overview

- Review: Heaps and MAX_HEAPIFY
- Building a Heap
- Heap Sort
- Priority Queues (Recitation)

Readings

CLRS 6.1-6.4

Review

Heaps:

$$\begin{aligned} \text{Parent}(i) &= \lfloor i/2 \rfloor \\ \text{Left}(i) &= 2i \\ \text{Right}(i) &= 2i+1 \end{aligned}$$

Max_heap property:

A[Parent(i)] $\geq$ A[i]

- MAX_HEAPIFY($A, 2$)
  heap_size(A) = 10
  $A[2] \longleftrightarrow A[4]$
- MAX_HEAPIFY(A,4)
  $A[4] \longleftrightarrow A[9]$

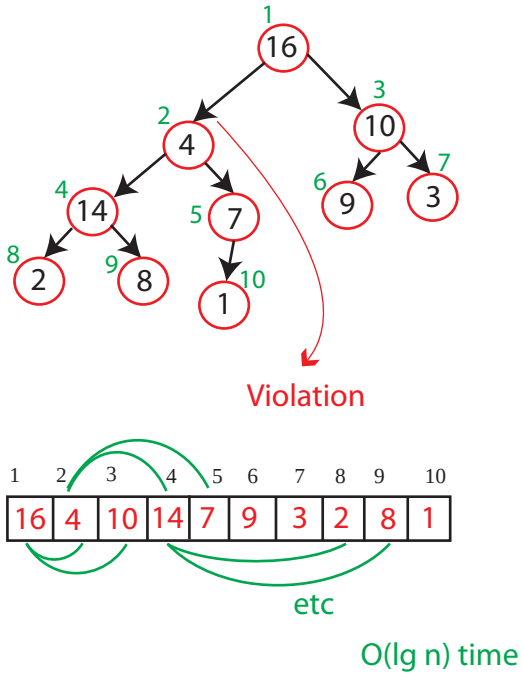


Figure 1: Review from last lecture

Building a Heap

$A[1 \cdots n]$ converted to a max_heap *Observation*: Elements $A[\lfloor n/2+1 \rfloor \cdots n]$ are all leaves of the tree and can't have children.

BUILD_MAX_HEAP(A):
heap_size(A) = length(A)
$O(n)$ times for $i \leftarrow \lfloor$ length$[A]/2 \rfloor$ downto 1
$O(\lg n)$ time do MAX_HEAPIFY(A, i)
$O(n \lg n)$ overall

See Figure 2 for an example.

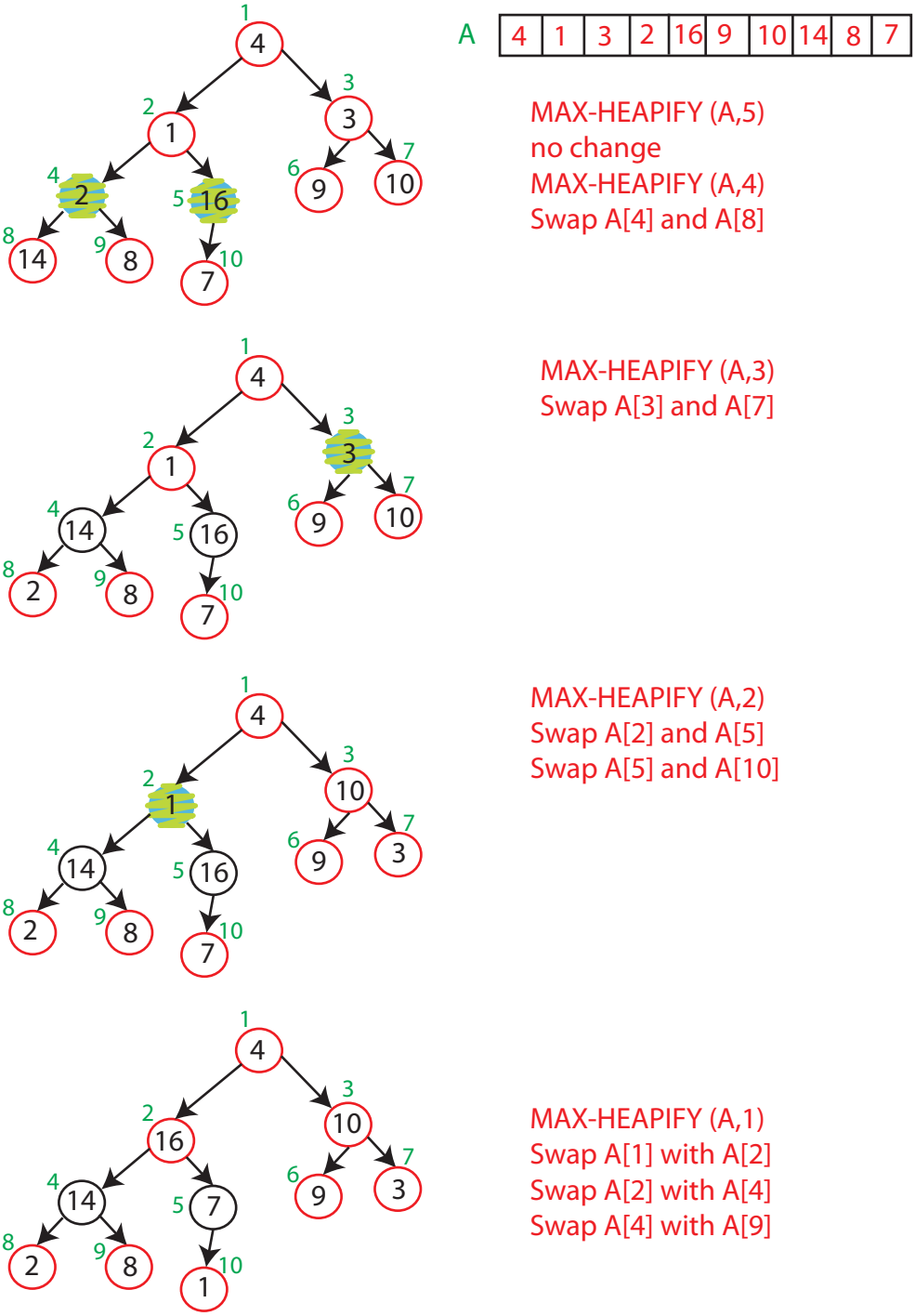


Figure 2: Example: Building Heaps

Sorting Strategy

- Build max_heap from unordered array
- Find maximum element ($A[1]$)
- Put it in correct position $A[n]$, $A[n]$ goes to $A[1]$

  New root could violate max_heap property but children remain max_heaps.
- Discard node n from heap (decrement heapsize)

Heap Sort Algorithm

$O(n \lg n)$	BUILD_MAX_HEAP(A):
n times	for i =length[A] downto 2
	do exchange $A[1] \longleftrightarrow A[i]$
	heap_size[A] = heap_size[A] $- 1$
$O(\lg n)$	MAX_HEAPIFY($A, 1$)
	$O(n \lg n)$ overall

See Figure 3 for an illustration.

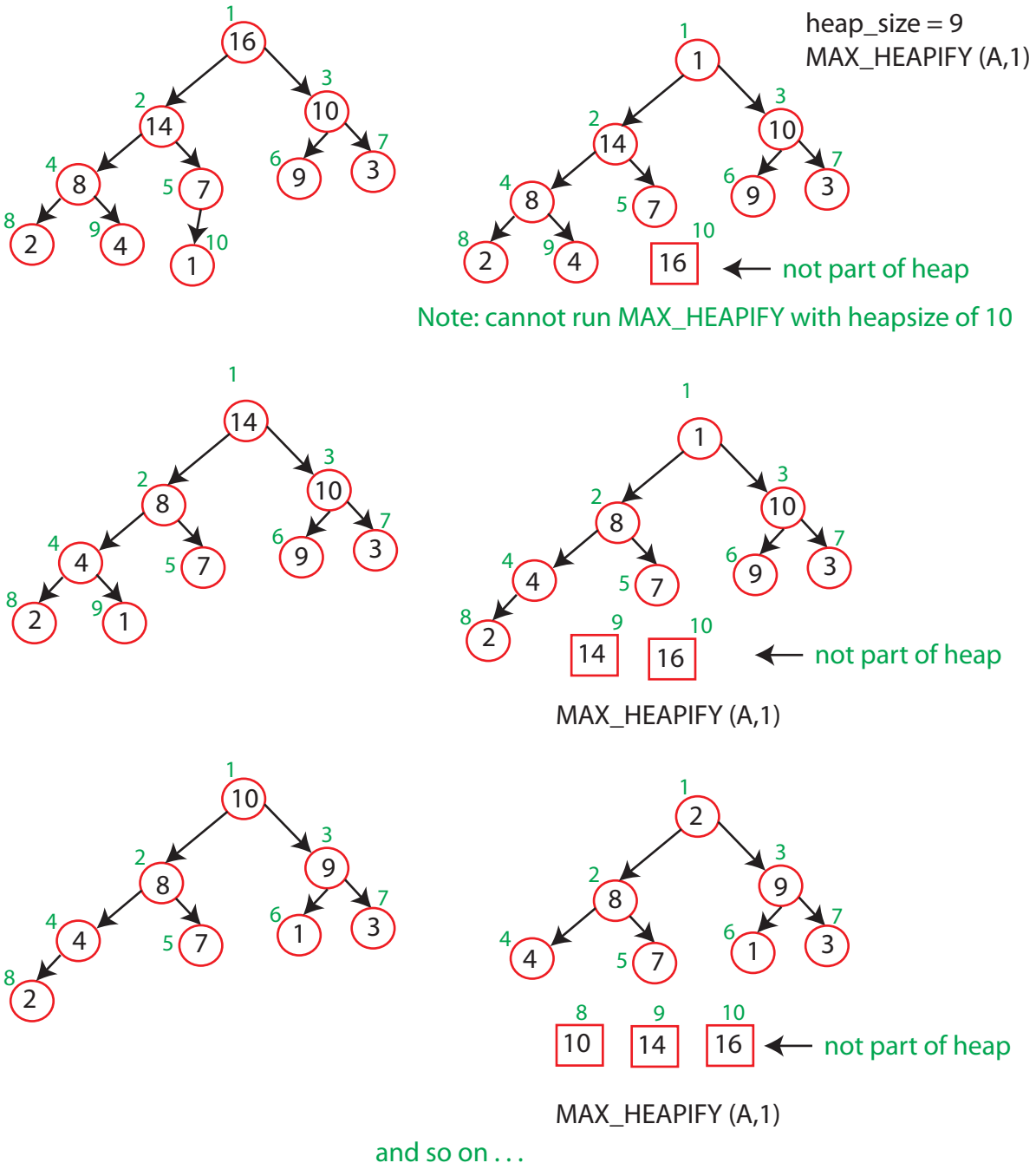


Figure 3: Illustration: Heap Sort Algorithm

Priority Queues

This is an abstract datatype as it can be implemented in different ways.

INSERT(S, X) :	inserts X into set S
MAXIMUM(S):	returns element of S with largest key
EXTRACT_MAX(S):	removes and returns element with largest key
INCREASE_KEY(S, x, k):	increases the value of element x's key to new value k (assumed to be as large as current value)

6.006 Introduction to Algorithms
Spring 2008

Lecture 10: Sorting III: Linear Bounds Linear-Time Sorting

Lecture Overview

- Sorting lower bounds
  - Decision Trees
- Linear-Time Sorting
  - Counting Sort

Readings

CLRS 8.1-8.4

Comparison Sorting

Insertion sort, merge sort and heap sort are all comparison sorts.
The best worst case running time we know is $O(n \lg n)$. Can we do better?

Decision-Tree Example

Sort $< a_1, a_2, \cdots a_n >$.

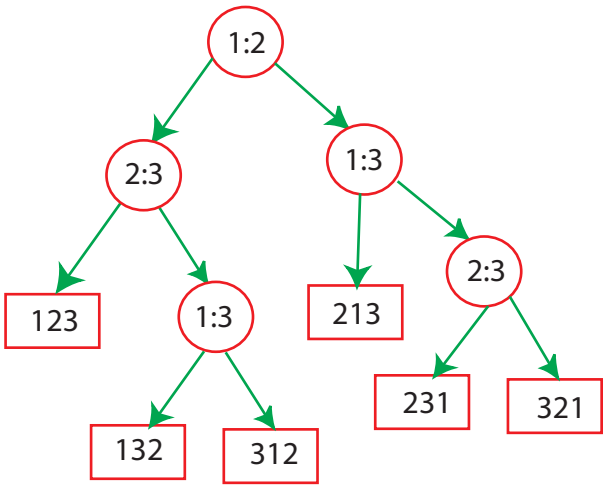


Figure 1: Decision Tree

Each internal node labeled $i : j$, compare a_i and a_j, go left if $a_i \leq a_j$, go right otherwise.

Example

Sort $< a_1, a_2, a_3 >=< 9, 4, 6 >$ Each leaf contains a permutation, i.e., a total ordering.

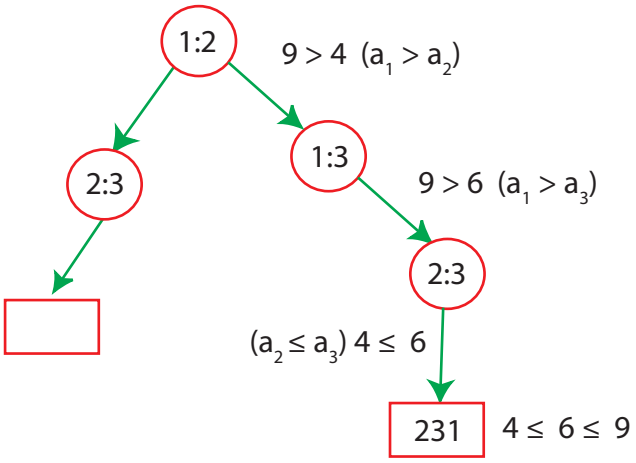


Figure 2: Decision Tree Execution

Decision Tree Model

Can model execution of any comparison sort. In order to sort, we need to generate a total ordering of elements.

- One tree size for each input size n
- Running time of algo: length of path taken
- Worst-case running time: height of the tree

Theorem

Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof: Tree must contain $\geq n!$ leaves since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus,

$$\begin{aligned} n! &\leq 2^h \\ \implies h &\geq \lg(n!) \quad (\geq \lg((\frac{n}{e})^n) \text{ Stirling}) \\ &\geq n \lg n - n \lg e \\ &= \Omega(n \lg n) \end{aligned}$$

Sorting in Linear Time

Counting Sort: no comparisons between elements

Input: $A[1 \ldots n]$ where $A[j] \in \{1, 2, \cdots, k\}$

Output: $B[1 \ldots n]$ sorted

Auxiliary Storage: $C[1 \ldots k]$

Intuition

Since elements are in the range $\{1, 2, \cdots, k\}$, imagine collecting all the j's such that $A[j] = 1$, then the j's such that $A[j] = 2$, etc.

Don't compare elements, so it is not a comparison sort!

$A[j]$'s index into appropriate positions.

Pseudo Code and Analysis

```
θ(k)  {  for i ← 1 to k
            do C [i] = 0
θ(n)  {  for j ← 1 to n
            do C [A[j]] = C [A[j]] + 1
θ(k)  {  for i ← 2 to k
            do C [i] = C [i] + C [i-1]
θ(n)  {  for j ← n downto 1
            do B[C [A[j]]] = A[j]
               C [A[j]] = C [A[j]] - 1
-------------------------------------
θ(n+k)
```

Figure 3: Counting Sort

Example

Note: Records may be associated with the $A[i]$'s.

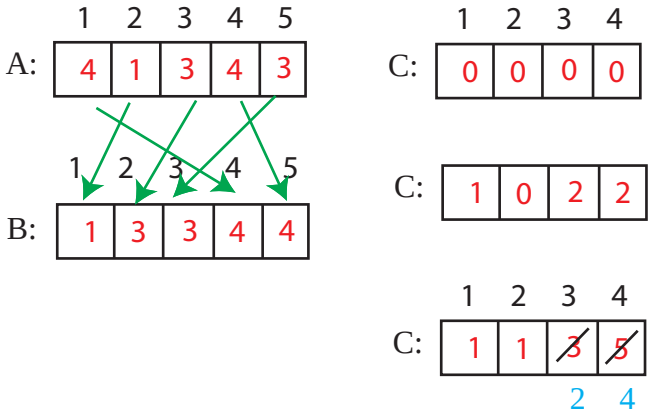


Figure 4: Counting Sort Execution

$$
\begin{aligned}
A[n] &= A[5] = 3 \\
C[3] &= 3 \\
B[3] &= A[5] = 3, C[3] \text{ decr.} \\
A[4] &= 4 \\
C[4] &= 5 \\
B[5] &= A[4] = 4, C[4] \text{ decr.} \quad \text{and so on } \ldots
\end{aligned}
$$

6.006 Introduction to Algorithms
Spring 2008

Lecture 11: Sorting IV: Stable Sorting, Radix Sort

Lecture Overview

- Stable Sorting
- Radix Sort
- Quick Sort ← not officially a part of 6.006
- Sorting Races

Stable Sorting

Preserves input order among equal elements

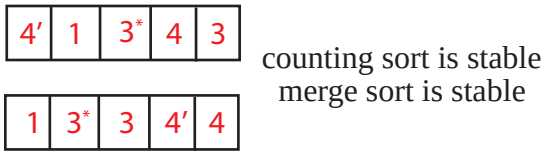


Figure 1: Stability

Selection Sort and Heap: Find maximum element and put it at end of array (swap with element at end of array) NOT STABLE!

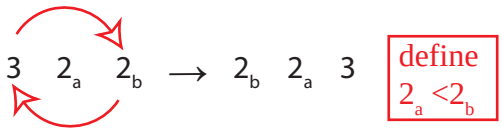


Figure 2: Selection Sort Instability

Radix Sort

- Herman Hollerith card-sorting machine for 1890 census.
- Digit by Digit sort by mechanical machine
  1. Examine given column of each card in a deck
  2. Distribute the card into one of 10 bins

3. Gather cards bin by bin, so cards with first place punched are on top of cards with second place punched, etc.

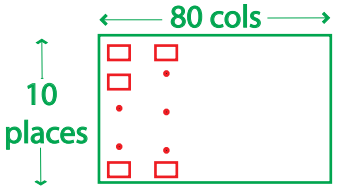


Figure 3: Punch Card

MSB vs. LSB?

Sort on most significant digit first or least significant digit first?

MSB strategy: Cards in 9 of 10 bins must be put aside, leading to a large number of intermediate piles

LSB strategy: Can gather sorted cards in bins appropriately to create a deck!

Example

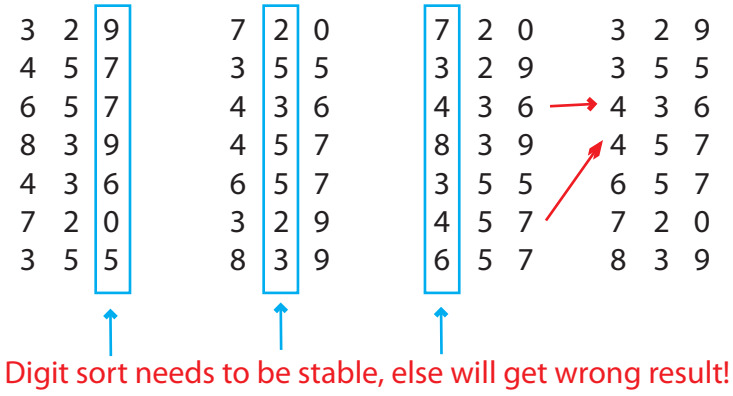


Figure 4: Example of Radix Sort

Analysis

Assume counting sort is auxiliary stable sort. $\Theta(n+k)$ complexity.

Suppose we have n words of b bits each.

One pass of counting sort	$\Theta(n+2^b)$		
b passes of counting sort	$\Theta(b(n+2)) = \Theta(nb)$		
$\frac{b}{r}$ passes	$\Theta(\frac{b}{r}(n+2^r))$	minimized when $r = \lg n$	$\Theta(\frac{bn}{\lg n})$

Quick Sort

This section is for "enrichment" only.

Divide: Partition the array into two. Sub-arrays around a pivot x such that elements in lower sub array $\leq x \leq$ elements in upper sub array. ← Linear Time

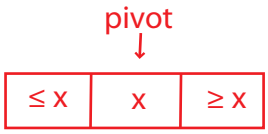


Figure 5: Pivot Definition

Conquer: Recursively sort the two sub arrays

Combine: Trivial

If we choose a pivot such that two sub arrays are roughly equal:

$$T(n) = 2T(n/2) + \Theta(n) \implies T(n) = \Theta(n \lg n)$$

If one array is much bigger:

$$T(n) = T(n-1) + \Theta(n) \implies T(n) = \Theta(n^2)$$

Average case $\Theta(n \lg n)$ assuming input array is randomized!

Sorting Races

Click here for a reference on this.

Bubble Sort: Repeatedly step through the list to be sorted. Compare 2 items, swap if they are in the wrong order. Continue through list, until no swaps. Repeat pass through list until no swaps. $\Theta(n^2)$

Shell Sort: Improves insertion sort by comparing elements separated by gaps $\Theta(n\lg^2 n)$

6.006 Introduction to Algorithms
Spring 2008

Lecture 12: Searching I: Graph Search and Representations

Lecture Overview: Search 1 of 3

- Graph Search
- Applications
- Graph Representations
- Introduction to breadth-first and depth-first search

Readings

CLRS 22.1-22.3, B.4

Graph Search

Explore a graph e.g., find a path from start vertices to a desired vertex
Recall: graph $G = (V, E)$

- V = set of vertices (arbitrary labels)
- E = set of edges i.e. vertex pairs (v, w)
  - ordered pair $\implies$ *directed* edge of graph
  - unordered pair $\implies$ *undirected*

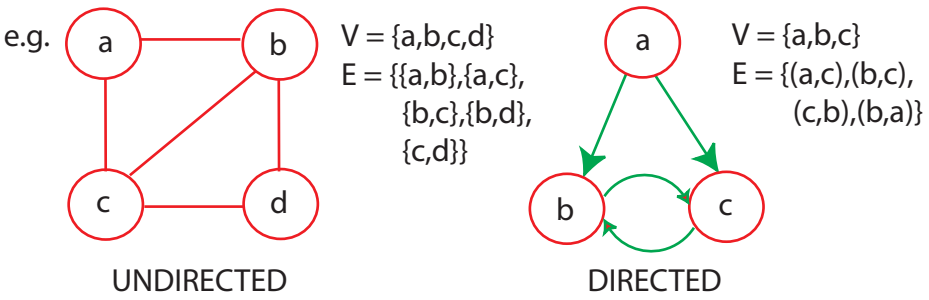


Figure 1: Example to illustrate graph terminology

Applications:

There are many.

- web crawling (How Google finds pages)
- social networking (Facebook friend finder)
- computer networks (Routing in the Internet)
  shortest paths [next unit]
- solving puzzles and games
- checking mathematical conjectures

Pocket Cube:

Consider a $2 \times 2 \times 2$ Rubik's cube

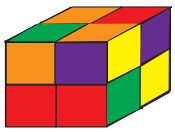


Figure 2: Rubik's Cube

- Configuration Graph:
  - vertex for each possible state
  - edge for each basic move (e.g., 90 degree turn) from one state to another
  - undirected: moves are reversible
- Puzzle: Given initial state s, find a path to the solved state
- $\sharp$ vertices $= 8!.3^8 = 264,539,520$ (because there are 8 cubelets in arbitrary positions, and each cubelet has 3 possible twists)

Figure 3: Illustration of Symmetry

- can factor out 24-fold symmetry of cube: fix one cubelet

$$8^{11}.3 \implies 7!.3^7 = 11,022,480$$

- in fact, graph has 3 connected components of equal size $\implies$ only need to search in one

$$\implies 7!.3^6 = 3,674,160$$

"Geography" of configuration graph

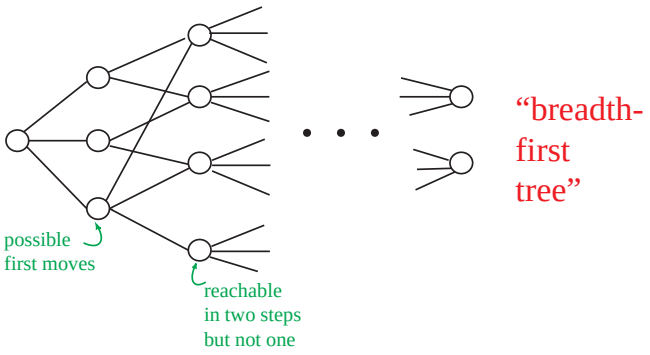


Figure 4: Breadth-First Tree

♯ reachable configurations

distance	90° turns	90° & 180° turns
0	1	1
1	6	9
2	27	54
3	120	321
4	534	1,847
5	2,256	9,992
6	8,969	50,136
7	33,058	227,536
8	114,149	870,072
9	360,508	1,887,748
10	930,588	623,800
11	1,350,852	2,644 ← diameter
12	782,536	
13	90,280	
14	276 ← diameter	
	3,674,160	3,674,160
		Wikipedia Pocket Cube

Cf. $3 \times 3 \times 3$ Rubik's cube: ≈ 1.4 trillion states; diameter is unknown! ≤ 26

Representing Graphs: (data structures)

Adjacency lists:

Array Adj of $|V|$ linked lists

- for each vertex $u \epsilon V$, $Adj[u]$ stores u's neighbors, i.e., $\{v \epsilon V \mid (u,v) \epsilon E\}$. colorBlue$(u,v)$ are just outgoing edges if directed. (See Fig. 5 for an example)
- in Python: Adj = dictionary of list/set values vertex = any hashable object (e.g., int, tuple)
- advantage: multiple graphs on same vertices

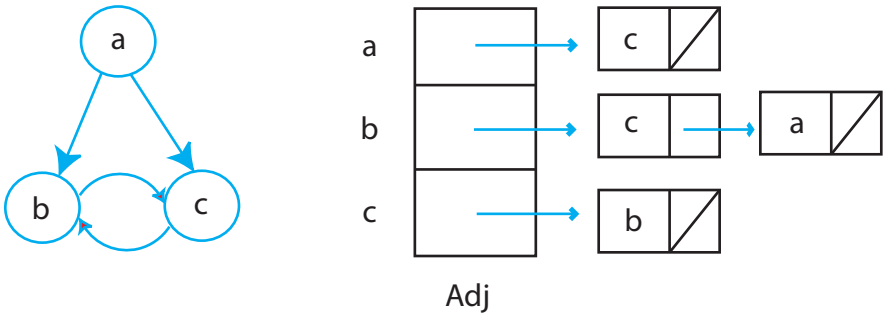


Figure 5: Adjacency List Representation

Object-oriented variations:

- object for each vertex u
- u.neighbors = list of neighbors i.e., $Adj[u]$

Incidence Lists:

- can also make edges objects (see Figure 6)
- u.edges = list of (outgoing) edges from u.
- advantage: storing data with vertices and edges without hashing

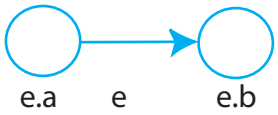


Figure 6: Edge Representation

Representing Graphs: contd.

The above representations are good for for sparse graphs where $| E | \ll (| V |)^2$. This translates to a space requirement $= \Theta(V + E)$ (Don't bother with $| . |$'s inside O/Θ).

Adjacency Matrix:

- assume $V = \{1, 2, \ldots, |v|\}$ (number vertices)
- $A = (a_{ij}) = |V| \times |V|$ matrix where $i =$ row and $j =$ column, and

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \ \epsilon \text{ E} \\ \phi & \text{otherwise} \end{cases}$$

  See Figure 7.
- good for dense graphs where $| E | \approx (| V |)^2$
- space requirement $= \Theta(V^2)$
- cool properties like A^2 gives length-2 paths and Google PageRank $\approx A^\infty$
- but we'll rarely use it Google couldn't; $| V | \approx$ 20 billion $\implies$ $(| V |)^2 \approx 4.10^{20}$ [50,000 petabytes]

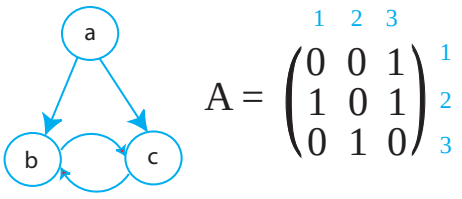


Figure 7: Matrix Representation

Implicit Graphs:

Adj(u) is a function or u.neighbors/edges is a method $\implies$ "no space" (just what you need now)

High level overview of next two lectures:

Breadth-first search

Levels like "geography"

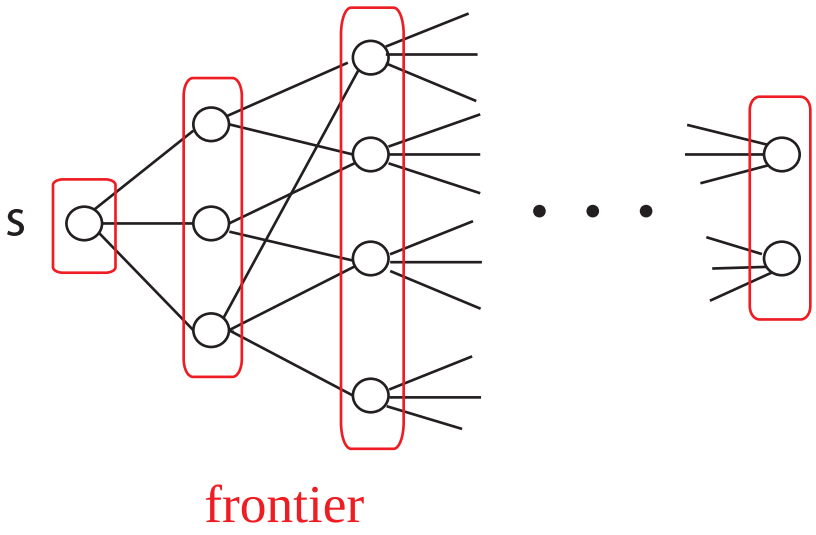


Figure 8: Illustrating Breadth-First Search

- frontier = current level
- initially $\{s\}$
- repeatedly advance frontier to next level, careful not to go backwards to previous level
- actually find shortest paths i.e. fewest possible edges

Depth-first search

This is like exploring a maze.

- e.g.: (left-hand rule) - See Figure 9
- follow path until you get stuck
- backtrack along breadcrumbs until you reach an unexplored edge

- recursively explore it
- careful not to repeat a vertex

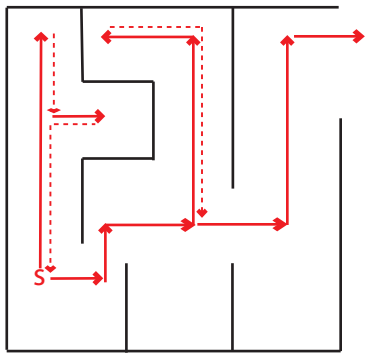


Figure 9: Illustrating Depth-First Search

6.006 Introduction to Algorithms
Spring 2008

Lecture 13: Searching II: Breadth-First Search and Depth-First Search

Lecture Overview: Search 2 of 3

- Breadth-First Search
- Shortest Paths
- Depth-First Search
- Edge Classification

Readings

CLRS 22.2-22.3

Recall:

graph search: explore a graph
e.g., find a path from start vertices to a desired vertex

adjacency lists: array Adj of $|V|$ linked lists

- for each vertex $u \epsilon V$, Adj$[u]$ stores u's neighbors, i.e. $\{v \epsilon V \mid (u, v) \epsilon E\}$
  v - just outgoing edges if directed

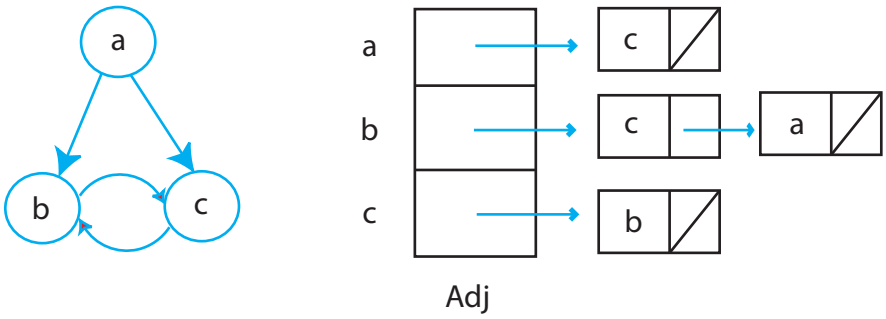


Figure 1: Adjacency Lists

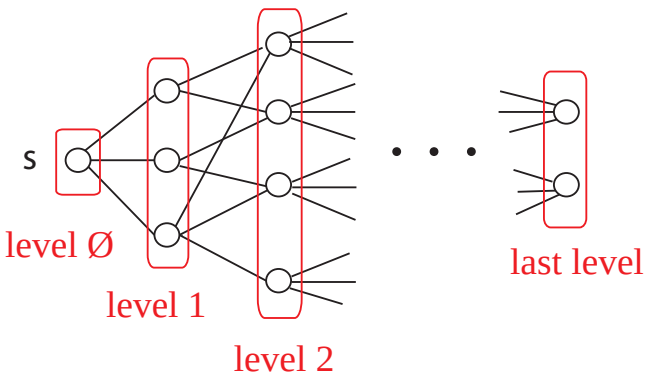


Figure 2: Breadth-First Search

Breadth-first Search (BFS):

See Figure 2
Explore graph level by level from S

- level $\phi = \{s\}$
- level i = vertices reachable by path of i edges but not fewer
- build level $i > 0$ from level $i-1$ by trying all outgoing edges, but ignoring vertices from previous levels

```
BFS (V,Adj,s):
    level = { s: φ }
    parent = {s : None }
    i = 1
    frontier = [s]                    ♯ previous level, i − 1
    while frontier:
        next = [ ]                    ♯ next level, i
        for u in frontier:
            for v in Adj[u]:
                if v not in level:    ♯ not yet seen
                    level[v] = i      ♯ = level[u] + 1
                    parent[v] = u
                    next.append(v)
        frontier = next
        i += 1
```

Example:

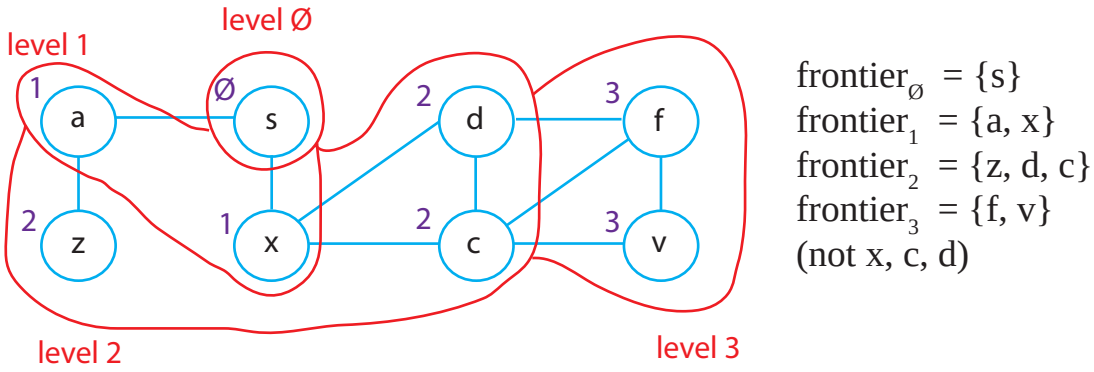


Figure 3: Breadth-First Search Frontier

Analysis:

- vertex V enters next (& then frontier)
  only once (because level[v] then set)
  base case: $v = s$
- $\Longrightarrow$ Adj[v] looped through only once

$$\text{time} = \sum_{v \epsilon V} \mid Adj[V] \mid = \begin{cases} \mid E \mid & \text{for directed graphs} \\ 2 \mid E \mid & \text{for undirected graphs} \end{cases}$$

- $O(E)$ time
  - $O(V + E)$ to also list vertices unreachable from v (those still not assigned level)
  "LINEAR TIME"

Shortest Paths:

- for every vertex v, fewest edges to get from s to v is

$$\begin{cases} \text{level}[v] \text{ if } v \text{ assigned level} \\ \infty \text{ else (no path)} \end{cases}$$

- parent pointers form shortest-path tree = union of such a shortest path for each v
  $\Longrightarrow$ to find shortest path, take v, parent[v], parent[parent[v]], etc., until s (or None)

Depth-First Search (DFS):

This is like exploring a maze.

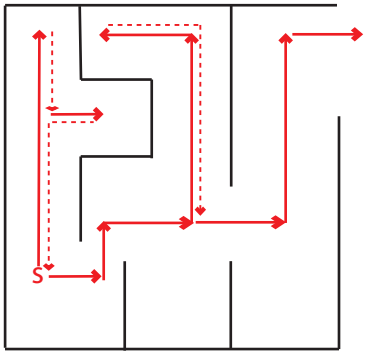


Figure 4: Depth-First Search Frontier

- follow path until you get stuck
- backtrack along breadcrumbs until reach unexplored neighbor
- recursively explore

```
parent  = {s: None}

DFS-visit (V, Adj, s):
   for v in Adj [s]:
      if v not in parent:
         parent [v] = s
         DFS-visit (V, Adj, v)

DFS (V, Adj)
   parent = { }
   for s in V:
      if s not in parent:
         parent [s] = None
         DFS-visit (V, Adj, s)
```

search from start vertex s (only see stuff reachable from s)

explore entire graph

(could do same to extend BFS)

Figure 5: Depth-First Search Algorithm

Example:

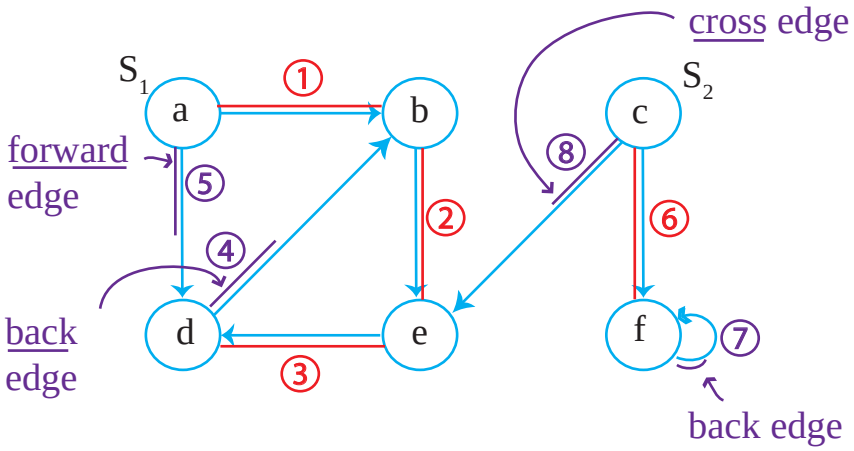


Figure 6: Depth-First Traversal

Edge Classification:

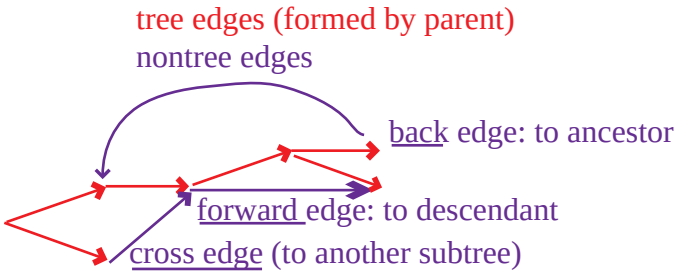


Figure 7: Edge Classification

To compute this classification, keep global time counter and store time interval during which each vertex is on recursion stack.

Analysis:

- DFS-visit gets called with a vertex s only once (because then parent[s] set)
  $\Longrightarrow$ time in DFS-visit $= \sum_{s \epsilon V} |\ \text{Adj}[s]\ | = O(E)$
- DFS outer loop adds just $O(V)$
  $\Longrightarrow$ $O(V + E)$ time (linear time)

6.006 Introduction to Algorithms
Spring 2008

Lecture 14: Searching III: Toplogical Sort and NP-completeness

Lecture Overview: Search 3 of 3 & NP-completeness

- BFS vs. DFS
- job scheduling
- topological sort
- intractable problems
- P, NP, NP-completeness

Readings

CLRS, Sections 22.4 and 34.1-34.3 (at a high level)

Recall:

- Breadth-First Search (BFS): level by level
- Depth-First Search (DFS): backtrack as necc.
- both $O(V + E)$ worst-case time $\implies$ optimal
- BFS computes shortest paths (min. ♯ edges)
- DFS is a bit simpler & has useful properties

Job Scheduling:

Given Directed Acylic Graph (DAG), where vertices represent tasks & edges represent dependencies, order tasks without violating dependencies

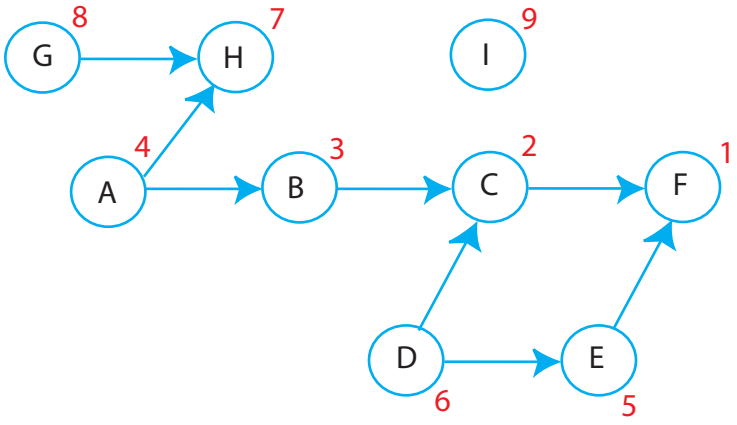


Figure 1: Dependence Graph

Source

Source = vertex with no incoming edges
= schedulable at beginning (A,G,I)

Attempt

BFS from each source:

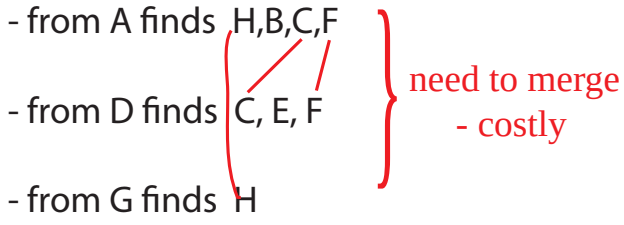


Figure 2: BFS-based Scheduling

Topological Sort

Reverse of DFS finishing times (time at which node's outgoing edges finished)
Exercise: prove that no constraints are violated

Intractability

- DFS & BFS are worst-case optimal if problem is really graph search (to look at graph)
- what if graph ...
  - is implicit?
  - has special structure?
  - is infinite?

The first 2 characteristics (implicitness and special structure) apply to the Rubik's Cube problem.
The third characteristic (infiniteness) applies to the Halting Problem.

Halting Problem:

Given a computer program, does it ever halt (stop)?

decision problem: answer is YES or NO

UNDECIDABLE: no algorithm solves this problem (correctly in finite time on all inputs)

Most decision problems are undecidable:

- program $\approx$ binary string $\approx$ nonneg. integer ϵ $\aleph$
- decision problem = a function from binary strings to {YES,NO}. Binary strings refer to $\approx$ nonneg. integers while {YES,NO} $\approx$ {0,1}
- $\approx$ infinite sequence of bits $\approx$ real number ϵ $\Re$
- $\aleph \ll \Re$: non assignment of unique nonneg. integers to real numbers ($\Re$ uncountable)
- $\Longrightarrow$ not nearly enough programs for all problems & each program solves only one problem
- $\Longrightarrow$ almost all problems cannot be solved

$n \times n \times n$ Rubik's cube:

- $n = 2$ or 3 is easy algorithmically: $O(1)$ time
  in practice, $n = 3$ still unsolved
- graph size grows exponentially with n
- solvability decision question is easy (parity check)
- finding shortest solution: UNSOLVED

$n \times n$ Chess:

Given $n \times n$ board & some configuration of pieces, can WHITE force a win?

- can be formulated as $(\alpha\beta)$ graph search
- every algorithm needs time exponential in n:
  "EXPTIME-complete" [Fraenkel & Lichtenstein 1981]

$n^2 - 1$ Puzzle:

Given $n \times n$ grid with $n^2 - 1$ pieces, sort pieces by sliding (see Figure 3).

- similar to Rubik's cube:
- solvability decision question is easy (parity check)
- finding shortest solution: NP-COMPLETE [Ratner & Warmuth 1990]

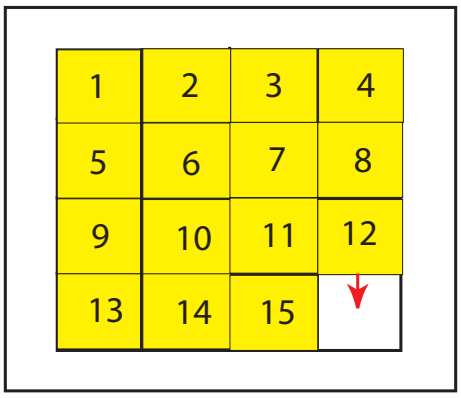


Figure 3: Puzzle

Tetris:

Given current board configuration & list of pieces to come, stay alive

- NP-COMPLETE [Demaine, Hohenberger, Liben-Nowell 2003]

P, NP, NP-completeness

P = all (decision) problems solvable by a polynomial ($O(n^c)$) time algorithm (efficient)

NP = all decision problems whose YES answers have short (polynomial-length) "proofs" checkable by a polynomial-time algorithm
e.g., Rubik's cube and $n^2 - 1$ puzzle:
is there a solution of length $\leq k$?
YES $\implies$ easy-to-check short proof(moves)
Tetris ϵ NP
but we conjecture Chess not NP (winning strategy is big- exponential in n)

P $\neq$ NP: Big conjecture (worth \$1,000,000) $\approx$ generating proofs/solutions is harder than checking them

NP-complete = in NP & NP-hard

NP-hard = as hard as every problem in NP
= every problem in NP can be efficiently converted into this problem
$\implies$ if this problem ϵ P then P = NP (so probably this problem not in P)

6.006 Introduction to Algorithms
Spring 2008

Lecture 15: Shortest Paths I: Intro

Lecture Overview

- Homework Preview
- Weighted Graphs
- General Approach
- Negative Edges
- Optimal Substructure

Readings

CLRS, Sections 24 (Intro)

Motivation:

Shortest way to drive from A to B (Google maps "get directions")

Formulation: Problem on a weighted graph $G(V, E)$ $W : E \to \Re$

Two algorithms: Dijkstra $O(V \lg V + E)$ assumes non-negative edge weights
Bellman Ford $O(VE)$ is a general algorithm

Problem Set 5 Preview:

- Use Dijkstra to find shortest path from CalTech to MIT
  - See "CalTech Cannon Hack" photos (search web.mit.edu)
  - See Google Maps from CalTech to MIT
- Model as a weighted graph $G(V, E), W : E \to \Re$
  - V = vertices (street intersections)
  - E = edges (street, roads); directed edges (one way roads)
  - $W(U, V)$ = weight of edge from u to v (distance, toll)

$$\text{path } p = < v_0, v_1, \ldots v_k >$$
$$(v_i, v_{i+1}) \, \epsilon \, E \quad \text{for} \quad 0 \leq i < k$$
$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Weighted Graphs:

Notation:

$v_0 \overset{p}{\longrightarrow} v_k$ means p is a path from v_0 to v_k. (v_0) is a path from v_0 to v_0 of weight 0.

Definition:

Shortest path weight from u to v as

$$\delta(u,v) = \begin{cases} \min \left\{ w(p) : u \overset{p}{\longrightarrow} v \right\} & \text{if } \exists \text{ any such path} \\ \infty & \text{otherwise} \quad (v \text{ unreachable from } u) \end{cases}$$

Single Source Shortest Paths:

Given $G = (V, E), w$ and a source vertex S, find $\delta(S, V)$ [and the best path] from S to each $v \epsilon V$.

Data structures:

$$\begin{aligned} d[v] &= \text{value inside circle} \\ &= \left\{ \begin{array}{ll} 0 & \text{if } v = s \\ \infty & \text{otherwise} \end{array} \right\} \Longleftarrow \quad \text{initially} \\ &= \delta(s,v) \Longleftarrow \quad \text{at end} \\ d[v] &\geq \delta(s,v) \quad \text{at all times} \end{aligned}$$

$d[v]$ decreases as we find better paths to v

$\Pi[v]$ = predecessor on best path to v, $\Pi[s]$ = NIL

Example:

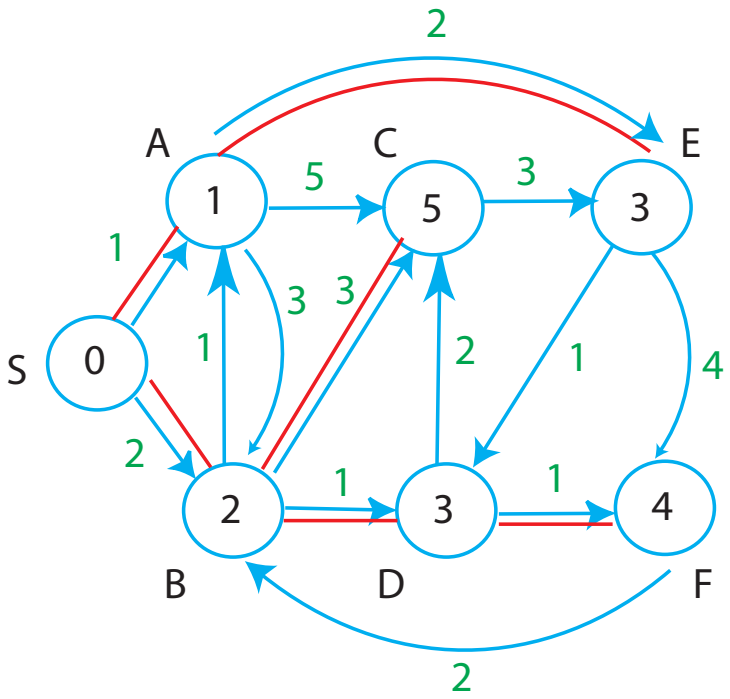


Figure 1: Shortest Path Example: Bold edges give predecessor Π relationships

Negative-Weight Edges:

- Natural in some applications (e.g., logarithms used for weights)
- Some algorithms disallow negative weight edges (e.g., Dijkstra)
- If you have negative weight edges, you might also have negative weight cycles $\Longrightarrow$ may make certain shortest paths undefined!

Example:

See Figure 2

$B \to D \to C \to B$ (origin) has weight $-6 + 2 + 3 = -1 < 0$!
Shortest path $S \longrightarrow C$ (or B, D, E) is undefined. Can go around $B \to D \to C$ as many times as you like
Shortest path $S \longrightarrow A$ is defined and has weight 2

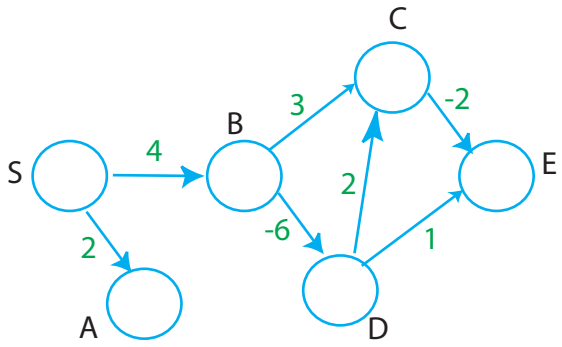


Figure 2: Negative-weight Edges

If negative weight edges are present, s.p. algorithm should find negative weight cycles (e.g., Bellman Ford)

General structure of S.P. Algorithms (no negative cycles)

Initialize: for $v \in V$: $d[v] \leftarrow \infty$, $\Pi[v] \leftarrow$ NIL

$d[S] \leftarrow 0$

Main: repeat

select edge (u, v) [somehow]

"Relax" edge (u, v):

if $d[v] > d[u] + w(u, v)$:

$d[v] \leftarrow d[u] + w(u, v)$

$\pi[v] \leftarrow u$

until all edges have $d[v] \leq d[u] + w(u, v)$

Complexity:

Termination? (needs to be shown even without negative cycles)
Could be exponential time with poor choice of edges.

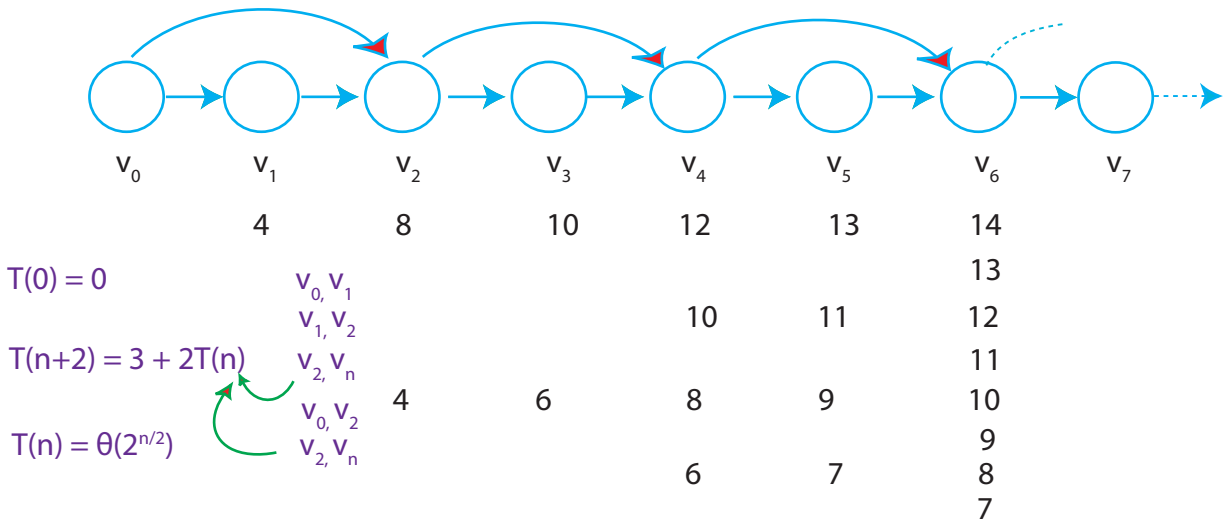


Figure 3: Running Generic Algorithm

Optimal Substructure:

Theorem: Subpaths of shortest paths are shortest paths

Let $p = < v_0, v_1, \ldots v_k >$ be a shortest path

Let $p_{ij} = < v_i, v_{i+1}, \ldots v_j > \quad 0 \leq i \leq j \leq k$

Then p_{ij} is a shortest path.

Proof:

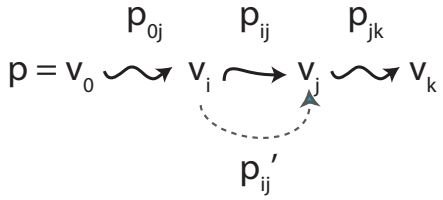


Figure 4: Optimal Substructure Theorem

If p'_{ij} is shorter than p_{ij}, cut out p_{ij} and replace with p'_{ij}; result is shorter than p.
Contradiction.

Triangle Inequality:

Theorem: For all $u, v, x \, \epsilon X$, we have

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v)$$

Proof:

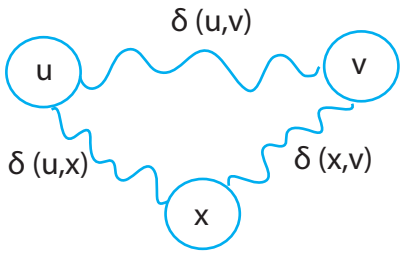


Figure 5: Triangle inequality

6.006 Introduction to Algorithms
Spring 2008

Lecture 16: Shortest Paths II: Bellman-Ford

Lecture Overview

- Review: Notation
- Generic S.P. Algorithm
- Bellman Ford Algorithm
  - Analysis
  - Correctness

Recall:

$$\begin{aligned} \text{path } p &= < v_0, v_1, \ \ldots, \ v_k > \\ & \quad (v_1, v_{i+1}) \ \epsilon E \quad 0 \leq i < k \\ w(p) &= \sum_{i-0}^{k-1} w(v_i, v_{i+1}) \end{aligned}$$

Shortest path weight from u to v is $\delta(u, v)$. $\delta(u, v)$ is ∞ if v is unreachable from u, undefined if there is a negative cycle on some path from u to v.

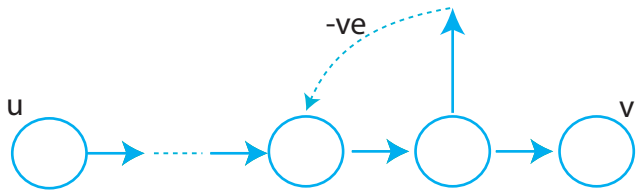


Figure 1: Negative Cycle

Generic S.P. Algorithm

Initialize:	for $v \ \epsilon \ V$: $\begin{array}{lll} d[v] & \leftarrow & \infty \\ \Pi[v] & \leftarrow & \text{NIL} \end{array}$
	$d[S] \leftarrow 0$
Main:	repeat
	select edge (u, v) [somehow]
"Relax" edge (u, v)	if $d[v] > d[u] + w(u, v)$: $d[v] \leftarrow d[u] + w(u, v)$ $\pi[v] \leftarrow u$
	until you can't relax any more edges or you're tired or ...

Complexity:

Termination: Algorithm will continually relax edges when there are negative cycles present.

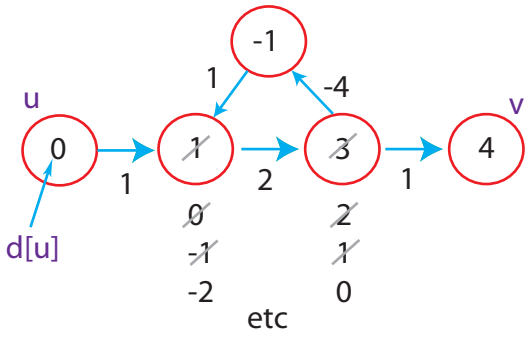


Figure 2: Algorithm may not terminate due to negative Cycles

Complexity could be exponential time with poor choice of edges.

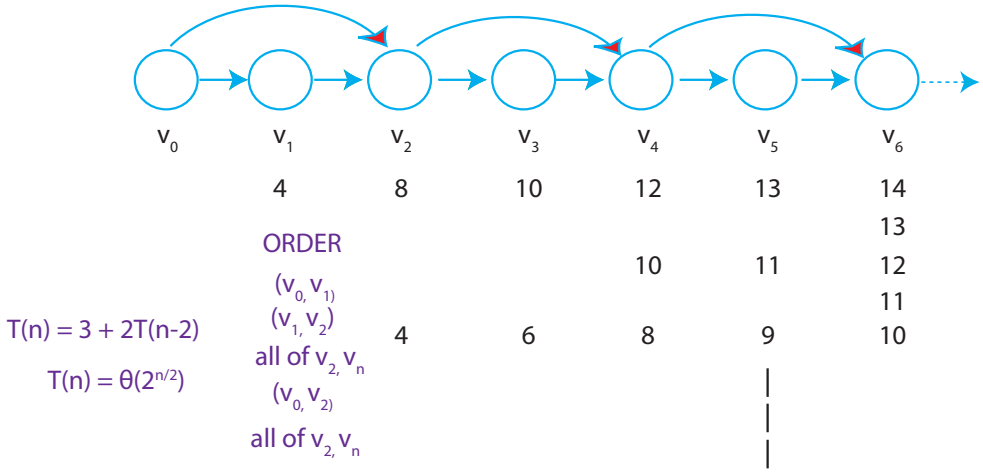


Figure 3: Algorithm could take exponential time

5-Minute 6.006

Here's what I want you to remember from 6.006 five years after you graduate

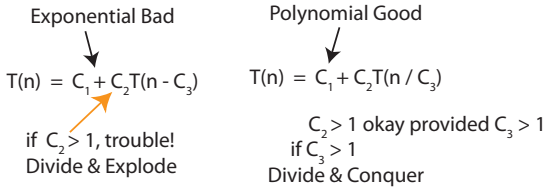


Figure 4: Exponential vs. Polynomial

Bellman-Ford(G,W,S)

Initialize ()
for $i = 1$ to $| v | -1$
    for each edge $(u, v) \epsilon E$:
        Relax(u, v)
for each edge $(u, v) \epsilon E$
    do if $d[v] > d[u] + w(u, v)$
        then report a negative-weight cycle exists

At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles

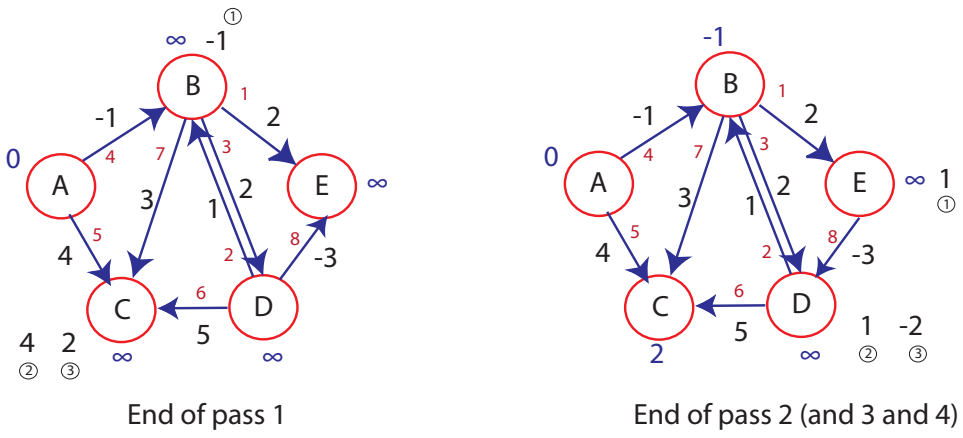


Figure 5: The numbers in circles indicate the order in which the δ values are computed

Theorem:
If $G = (V, E)$ contains no negative weight cycles, then after Bellman-Ford executes $d[v] = \delta(u, v)$ for all $v \epsilon V$.

Proof:
$v \epsilon V$ be any vertex. Consider path p from s to v that is a shortest path with minimum number of edges.

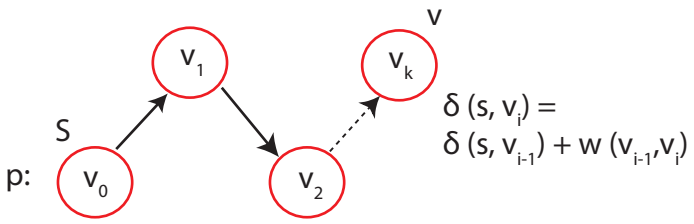


Figure 6: Illustration for proof

Initially $d[v_0] = 0 = \delta(S, V_0)$ and is unchanged since no negative cycles.
After 1 pass through E, we have $d[v_1] = \delta(s, v_1)$
After 2 passes through E, we have $d[v_2] = \delta(s, v_2)$
After k passes through E, we have $d[v_k] = \delta(s, v_k)$
No negative weight cycles $\implies$ p is simple $\implies$ p has $\leq \mid V \mid -1$ edges

Corollary

If a value $d[v]$ fails to converge after $\mid V \mid -1$ passes, there exists a negative-weight cycle reachable from s.

6.006 Introduction to Algorithms
Spring 2008

Lecture 17: Shortest Paths III - Dijkstra and Special Cases

Lecture Overview

- Shortest paths in DAGs
- Shortest paths in graphs without negative edges
- Dijkstra's Algorithm

Readings

CLRS, Sections 24.2-24.3

DAGs:

Can't have negative cycles because there are no cycles!

1. Topologically sort the DAG. Path from u to v implies that u is before v in the linear ordering
2. One pass over vehicles in topologically sorted order relaxing each edge that leaves each vertex
   $\Theta(V + E)$ time

Example:

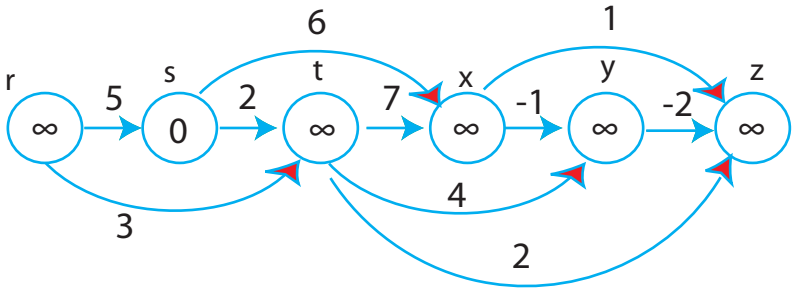


Figure 1: Shortest Path using Topological Sort

Vertices sorted left to right in topological order

Process r: stays ∞. All vertices to the left of s will be ∞ by definition

Process s: $t : \infty \to 2$ $x : \infty \to 6$ (see top of Figure 2)

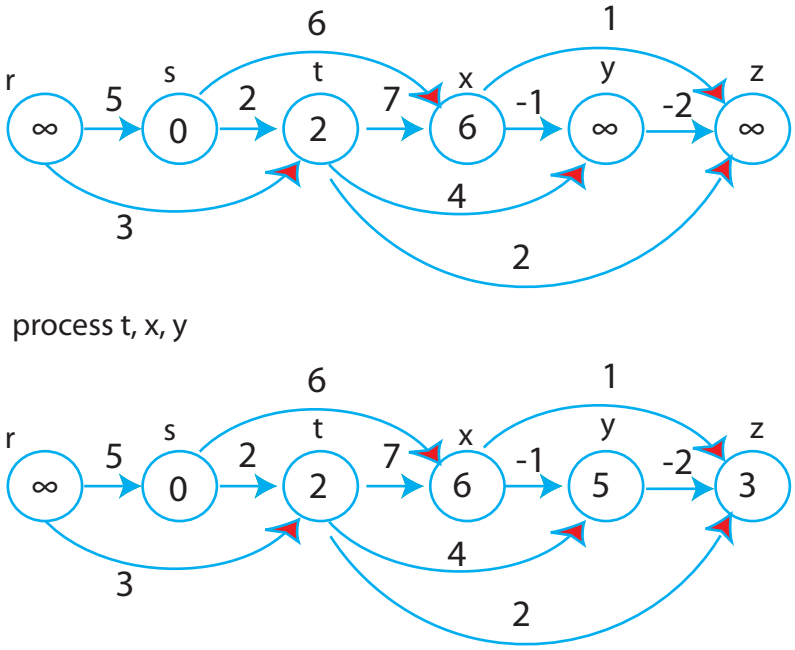


Figure 2: Preview of Dynamic Programming

Dijkstra's Algorithm

For each edge (u, v) ϵ E, assume $w(u, v) \geq 0$, maintain a set S of vertices whose final shortest path weights have been determined. Repeatedly select $u \epsilon V - S$ with minimum shortest path estimate, add u to S, relax all edges out of u.

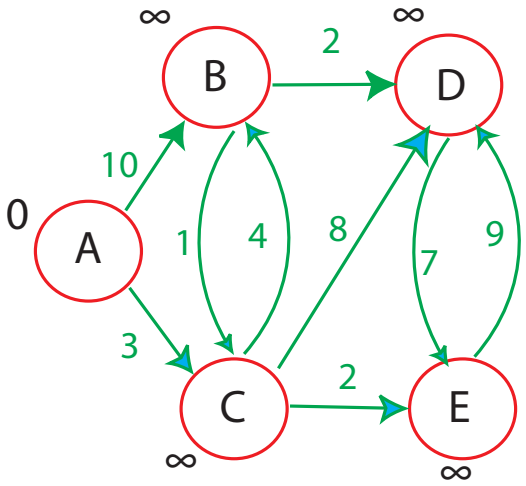
Pseudo-code

Dijkstra (G, W, s) //uses priority queue Q
    Initialize (G, s)
    $S \leftarrow \phi$
    $Q \leftarrow V[G]$ //Insert into Q
    while $Q \neq \phi$
        do $u \leftarrow$ EXTRACT-MIN(Q) //deletes u from Q
        $S = S \cup \{u\}$
        for each vertex $v \epsilon$ Adj$[u]$
            do RELAX (u, v, w) $\leftarrow$ this is an implicit DECREASE_KEY operation

Recall

$$\begin{aligned}&\text{RELAX}(u, v, w)\\ &\quad \text{if } d[v] > d[u] + w(u, v)\\ &\quad\quad \text{then } d[v] \leftarrow d[u] + w(u, v)\\ &\quad\quad\quad \text{TT}[v] \leftarrow u\end{aligned}$$

Example



S = { }	{ A B C D E } =	Q
S = { A }	0 ∞ ∞ ∞ ∞	
S = { A, C }	0 10 3 ∞ ∞	← after relaxing edges from A
S = { A, C }	0 7 3 11 5	← after relaxing edges from C
S = { A, C, E }	0 7 3 11 5	
S = { A, C , E, B}	0 7 3 9 5	← after relaxing edges from B

Figure 3: Dijkstra Execution

Strategy: Dijkstra is a greedy algorithm: choose closest vertex in $V - S$ to add to set S

Correctness: Each time a vertex u is added to set S, we have $d[u] = \delta(s, u)$

Complexity

$\theta(v)$ inserts into priority queue
$\theta(v)$ EXTRACT_MIN operations
$\theta(E)$ DECREASE_KEY operations

Array impl:

$\theta(v)$ time for extra min
$\theta(1)$ for decrease key
Total: $\theta(V.V + E.1) = \theta(V^2 + E) = \theta(V^2)$

Binary min-heap:

$\theta(\lg V)$ for extract min
$\theta(\lg V)$ for decrease key
Total: $\theta(V \lg V + E \lg V)$

Fibonacci heap (not covered in 6.006):

$\theta(\lg V)$ for extract min
$\theta(1)$ for decrease key
amortized cost
Total: $\theta(V \lg V + E)$

6.006 Introduction to Algorithms
Spring 2008

Lecture 18: Shortest Paths IV - Speeding up Dijkstra

Lecture Overview

- Single-source single-target Dijkstra
- Bidirectional search
- Goal directed search - potentials and landmarks

Readings

Wagner, Dorothea, and Thomas Willhalm. "Speed-Up Techniques for Shortest-Path Computations." In *Lecture Notes in Computer Science: Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Science*. Berlin/Heidelberg, MA: Springer, 2007. ISBN: 9783540709176. Read up to section 3.2.

DIJKSTRA single-source, single-target

```
Initialize()
Q ← V[G]
while Q ≠ φ
      do u ← EXTRACT_MIN(Q) (stop if u = t!)
      for each vertex v ϵ Adj[u]
          do RELAX(u, v, w)
```

Observation: If only shortest path from s to t is required, stop when t is removed from Q, i.e., when $u = t$

DIJKSTRA Demo

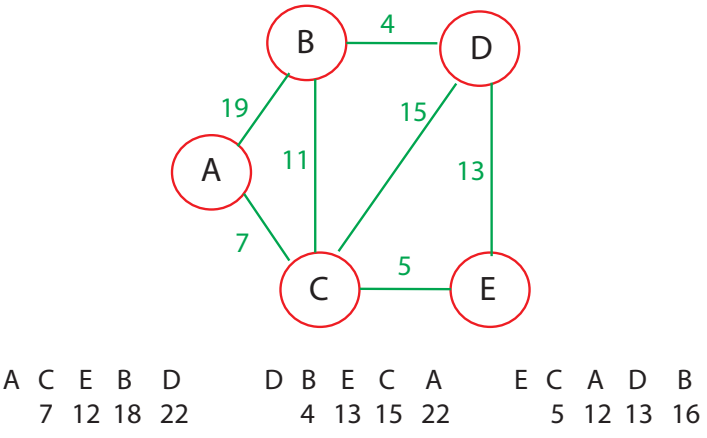


Figure 1: Dijkstra Demonstration with Balls and String

Bi-Directional Search

Note: Speedup techniques covered here do not change worst-case behavior, but reduce the number of visited vertices in practice.

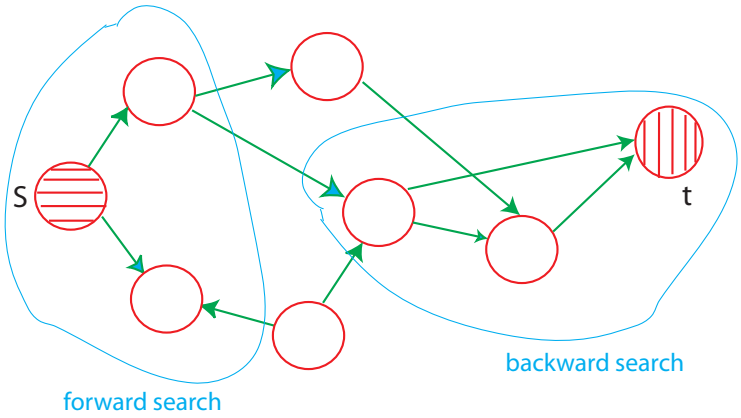


Figure 2: Bi-directional Search

Bi-D Search

Alternate forward search from s
backward search from t
(follow edges backward)
$d_f(u)$ distances for forward search
$d_b(u)$ distances for backward search

Algorithm terminates when some vertex w has been processed, i.e., deleted from the queue of both searches, Q_f and Q_b

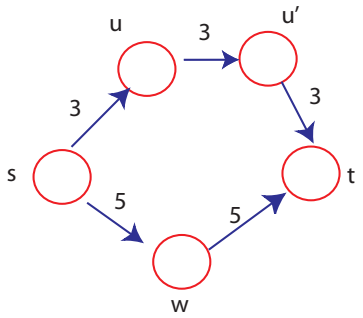


Figure 3: Bi-D Search

Subtlety: After search terminates, find node x with minimum value of $d_f(x) + d_b(x)$. x may not be the vertex w that caused termination as in example to the left!
Find shortest path from s to x using Π_f and shortest path backwards from t to x using Π_b.
Note: x will have been deleted from either Q_f or Q_b or both.

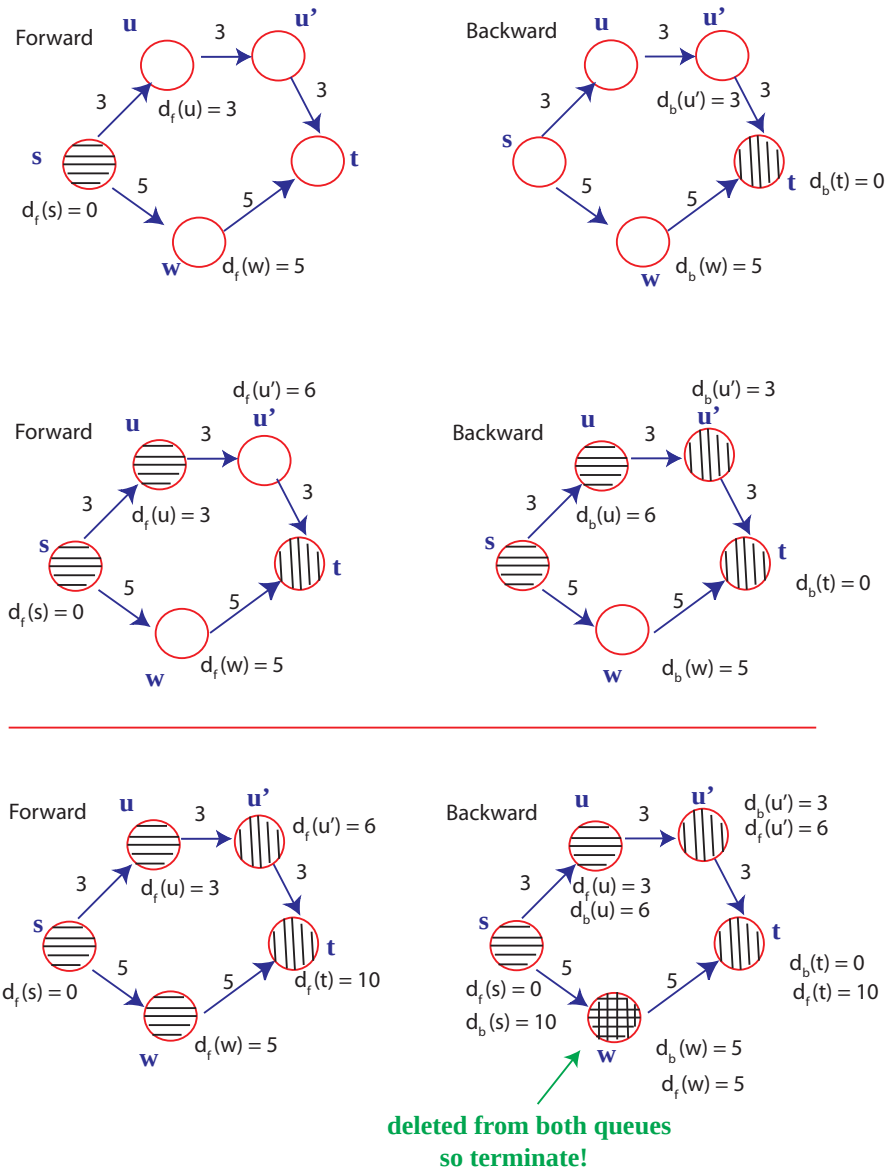


Figure 4: Forward and Backward Search

Minimum value for $d_f(x) + d_b(x)$ over all vertices that have been processed in at least one search

$$d_f(u) + d_b(u) = 3 + 6 = 9$$

$$d_f(u') + d_b(u') = 6 + 3 = 9$$
$$d_f(w) + d_b(w) = 5 + 5 = 10$$

Goal-Directed Search or A^*

Modify edge weights with potential function over vertices.

$$\overline{w}\ (u, v) = w\ (u, v) - \lambda(u) + \lambda(v)$$

Search toward target:

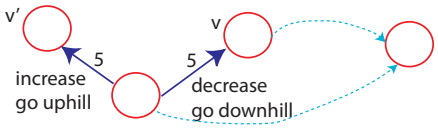


Figure 5: Targeted Search

Correctness

$$\overline{w}(p) = w(p) - \lambda_t(s) + \lambda_t(t)$$

So shortest paths are maintained in modified graph with $\overline{w}$ weights.

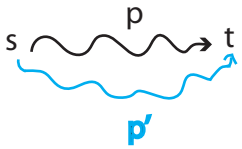


Figure 6: Modifying Edge Weights

To apply Dijkstra, we need $\overline{w}(u, v) \geq 0$ for all (u, v).
Choose potential function appropriately, to be feasible.

Landmarks

Small set of landmarks LCV. For all $u \epsilon V, l \epsilon L$, pre-compute $\delta(u, l)$. Potential $\lambda_t^{(l)}(u) = \delta(u, l) = \delta(t, l)$ for each l.
CLAIM: $\lambda_t^{(l)}$ is feasible.

Feasibility

$$\begin{aligned}
\overline{w}(u,v) &= w(u,v) - \lambda_t^{(l)}(u) + \lambda_t^{(l)}(v) \\
&= w(u,v) - \delta(u,l) + \delta(t,l) + \delta(v,l) - \delta(t,l) \\
&= w(u,v) - \delta(u,l) + \delta(v,l) \geq 0 \quad \text{by the } \Delta \text{ -inequality} \\
\lambda_t(u) &= \max_{l \,\epsilon\, L} \lambda_t^{(l)}(u) \;\; \text{is also feasible}
\end{aligned}$$

6.006 Introduction to Algorithms
Spring 2008

Lecture 19: Dynamic Programming I: Memoization, Fibonacci, Crazy Eights, Guessing

Lecture Overview

- Fibonacci Warmup
- Memoization and subproblems
- Shortest Paths
- Crazy Eights
- Guessing Viewpoint

Readings

CLRS 15

Dynamic Programming (DP)

Big idea: :hard yet simple

- Powerful algorithmic design technique
- Large class of seemingly exponential problems have a polynomial solution ("only") via DP
- Particularly for optimization problems (min / max) (e.g., shortest paths)

* DP $\approx$ "controlled brute force"
* DP $\approx$ recursion + re-use

Fibonacci Numbers

$$F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2}$$

Naive Algorithm

follow recursive definition

$\underline{\text{fib}}(n)$:
    if $n \leq 2$: return 1
    else return $\text{fib}(n-1) + \text{fib}(n-2)$
$\implies T(n) = T(n-1) + T(n-2) + O(1) \approx \phi^n$
    $\geq 2T(n-2) + O(1) \geq 2^{n/2}$
    EXPONENTIAL - BAD!

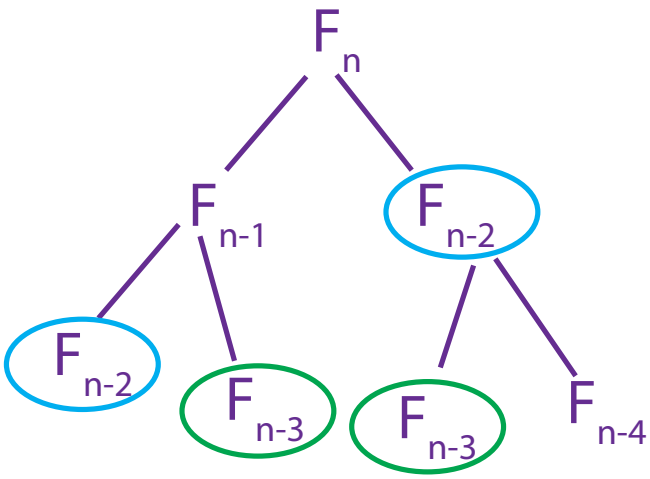


Figure 1: Naive Fibonacci Algorithm

Simple Idea

memoize

```
memo = { }
fib(n):
    if n in memo: return memo[n]
    else: if n ≤ 2 : f = 1
        else: f = fib(n − 1) + fib(n − 2)
                                 free
        memo[n] = f
        return f
```

$T(n) = T(n-1) + O(1) = O(n)$

[*Side Note*: There is also an $O(\lg n)$- time algorithm for Fibonacci, via different techniques]

* DP ≈ recursion + memoization
  - remember (memoize) previously solved "subproblems" that make up problem
    - in Fibonacci, subproblems are $F_0, F_1, \cdots, F_n$
  - if subproblem already solved, re-use solution
* $\implies$ time = ♯ of subproblems · time/subproblem
  - in fib: ♯ of subproblems is $O(n)$ and time/subproblem is $O(1)$ - giving us a total time of $O(n)$.

Shortest Paths

- Recursive formulation:
  $\delta(s,t) = \min\{w(s,v) + \delta(v,t) \,\big|\, (s,v) \in E\}$
- does this work with memoization?
  no, cycles $\implies$ infinite loops (see Figure 2).

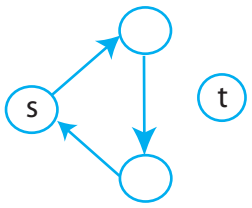


Figure 2: Shortest Paths

- in some sense necessary for neg-weight cycles
- works for directed acyclic graphs in $O(V+E)$
  (recursion effectively DFS/topological sort)
- trick for shortest paths: removing cyclic dependency.
  - $\delta_k(s,t)$ = shortest path using $\leq k$ edges
    $= \min\{\delta_{k-1}(s,t)\} \cup \{w(s,v) + \delta_{k-1}(v,t) \,\big|\, (s,v) \in E\}$
    ... except $\delta_k(t,t) = \phi$, $\delta_\phi(s,t) = \infty$ if $s \neq t$
  - $\delta(s,t) = \delta_{n-1}(s,t)$ assuming no negative cycles

$$\implies \text{time} = \underbrace{\sharp\text{subproblems}}_{O(n^3)\text{ for }s,t,k\cdots\text{ really }O(n^2)} \cdot \underbrace{\text{time/subproblem}}_{O(n)\cdots\text{ really deg}V}$$
$$= O(V \cdot \sum_V \deg(V)) = O(VE)$$

* Subproblem dependency should be acyclic.

Crazy Eights Puzzle

- given a sequence of cards $c[\phi], c[1], \cdots, c[n-1]$
  e.g., $7\heartsuit, 6\heartsuit, 7\diamondsuit, 3\diamondsuit, 8\clubsuit, J\spadesuit$
- find longest left-to-right "trick" (subsequence)

  $c[i_1], c[i_2], \cdots c[i_k] \quad (i_1 < i_2 < \cdots i_k)$
  where $c[i_j]$ & $c[i_{j+1}]$ "match" for all j
  have some suit or rank or one has rank 8

- recursive formulation:

  trick(i) = length of best trick starting at $c[i]$
  $= 1 + \max(\text{trick}(j) \text{ for } j \text{ in range}(i+1, n) \text{ if match } (c[i], c[j]))$
  best $= \max(\text{trick}(i) \text{ for } i \text{ in range}(n))$

- memoize: trick(i) depends only on trick($> i$)

  $\implies$ time $= \underbrace{\sharp \text{subproblems}}_{O(n)} \cdot \underbrace{\text{time/subproblem}}_{O(n)}$
  $= O(n^2)$ (to find actual trick, trace through max's)

"Guessing" Viewpoint

- what is the first card in best trick? guess!
  i.e., try all possibilities & take best result
  - only $O(n)$ choices
- what is next card in best trick from i? guess!
  - if you pretend you knew, solution becomes easy (using other subproblems)
  - actually pay factor of $O(n)$ to try all
- * use only small $\underbrace{\sharp \text{choices/guesses}}_{\text{poly}(n) \sim O(1)}$ per subproblem

6.006 Introduction to Algorithms
Spring 2008

Lecture 20: Dynamic Programming II: Longest Common Subsequence, Parent Pointers

Lecture Overview

- Review of big ideas & examples so far
- Bottom-up implementation
- Longest common subsequence
- Parent pointers for guesses

Readings

CLRS 15

Summary

* DP $\approx$ "controlled brute force"
* DP $\approx$ guessing + recursion + memoization
* DP $\approx$ dividing into reasonable $\sharp$ subproblems whose solutions relate - acyclicly - usually via guessing parts of solution.
* time = $\sharp$ subproblems $\times \underbrace{\text{time/subproblem}}_{\substack{\text{treating recursive calls as } O(1) \\ \text{(usually mainly guessing)}}}$
  - essentially an amortization
  - count each subproblem only once; after first time, costs $O(1)$ via memoization

Examples:	Fibonacci	Shortest Paths	Crazy Eights
subprobs:	fib(k)	$\delta_k(s,t) \forall s, k < n$	trick(i) = longest
	$0 \le k \le n$	= min path $s \to t$	trick from card(i)
		using $\le k$ edges	
♯ subprobs:	$\Theta(n)$	$\Theta(V^2)$	$\Theta(n)$
guessing:	none	edge from s, if any	next card j
♯ choices:	1	$\deg(s)$	$n - i$
relation:	= fib($k-1$)	$= \min\{\delta_{k-1}(s,t)\}$	$= 1 + \max(\text{trick}(j))$
	+ fib($k-2$)	$u\{w(s,v) + \delta_{k-1}(v,t)$	for $i < j < n$ if
		$\mid v \epsilon$ Adj$[s]\}$	match($c[i], c[j]$)
time/subpr:	$\Theta(1)$	$\Theta(1 + \deg(s))$	$\Theta(n - i)$
DP time:	$\Theta(n)$	$\Theta(VE)$	$\Theta(n^2)$
orig. prob:	fib(n)	$\delta_{n-1}(s,t)$	$\max\{\text{trick}(i), 0 \le i < n\}$
extra time:	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

Bottom-up implementation of DP:

alternative to recursion

- subproblem dependencies form DAG (see Figure 1)
- imagine topological sort
- iterate through subproblems in that order
  $\Longrightarrow$ when solving a subproblem, have already solved all dependencies
- often just: "solve smaller subproblems first"

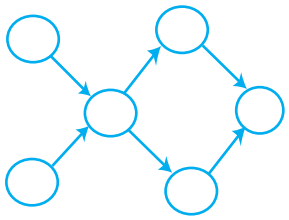


Figure 1: DAG

Example.

Fibonacci:
    for k in range($n+1$): fib$[k] = \cdots$
Shortest Paths:
    for k in range(n): for v in V : $d[k, v, t] = \cdots$
Crazy Eights:
    for i in reversed(range(n)): trick$[i] = \cdots$

- no recursion for memoized tests
  $\Longrightarrow$ faster in practice
- building DP table of solutions to all subprobs. can often optimize space:
  - Fibonacci: PS6
  - Shortest Paths: re-use same table $\forall k$

Longest common subsequence: (LCS)

A.K.A. edit distance, diff, CVS/SVN, spellchecking, DNA comparison, plagiarism, detection, etc.
Given two strings/sequences x & y, find longest common subsequence LCS(x,y) sequential but not necessarily contiguous

- e.g., H I E R O G L Y P H O L O G Y vs. M I C H A E L A N G E L O
  common subsequence is Hello
- equivalent to "edit distance" (unit costs): ♯ character insertions/deletions to transform $x \rightarrow y$ everything except the matches
- brute force: try all $2^{|x|}$ subsequences of $x \Longrightarrow \Theta(2^{|x|} \cdot |\ y\ |)$ time
- instead: DP on two sequences simultaneously

* Useful subproblems for strings/sequences x:
  - suffixes $x[i:]$
  - prefixes $x[:i]$
    The suffixes and prefixes are $\Theta(|\ x\ |) \longleftarrow\Longrightarrow$ use if possible
  - substrings $x[i:j]$ $\Theta(|\ x\ |^2)$

Idea: Combine such subproblems for x & y (suffixes and prefixes work)

LCS DP

- subproblem $c(i,j) = |\ \text{LCS}(x[i:], y[j:])\ |$ for $0 \le i, j < n$
  $\Longrightarrow \Theta(n^2)$ subproblems
  - original problem $\approx c[0,0]$ (length $\sim$ find seq. later)
- idea: either $x[i] = y[j]$ part of LCS or not $\Longrightarrow$ either $x[i]$ or $y[j]$ (or both) not in LCS (with anyone)
- guess: drop $x[i]$ or $y[j]$? (2 choices)

- relation among subproblems:

$$\text{if } x[i] = y[j] : c(i,j) = 1 + c(i+1, j+1)$$
$$\text{(otherwise } x[i] \text{ or } y[j] \text{ unused} \sim \text{can't help)}$$
$$\text{else: } c(i,j) = \max\{\underbrace{c(i+1,j)}_{x[i]\,\text{out}}, \underbrace{c(i,j+1)}_{y[j]\,\text{out}}\}$$
$$\text{base cases: } c(|\,x\,|, j) = c(i, |\,y\,|) = \phi$$
$$\implies \Theta(1) \text{ time per subproblem}$$
$$\implies \Theta(n^2) \text{ total time for DP}$$

- DP table: see Figure 2

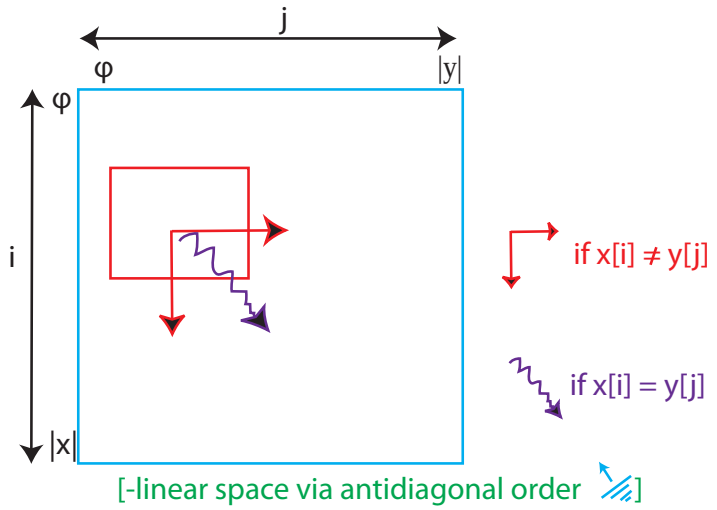


Figure 2: DP Table

- recursive DP:

```
def LCS(x, y):
    seen = { }
    def c(i, j):
        if i ≥ len(x) or j ≥ len(y) : return φ
        if (i, j) not in seen:
            if x[i] == y[j]:
                seen[i, j] = 1 + c(i + 1, j + 1)
            else:
                seen[i, j] = max(c(i + 1, j), c(i, j + 1))
        return seen[i, j]
    return c(0, 0)
```

- bottom-up DP:

```
def LCS(x, y):
    c = {}
    for i in range(len(x)):
        c[i, len(y)] = φ
    for j in range(len(y)):
        c[len(x), j] = φ
    for i in reversed(range(len(x))):
        for j in reversed(range(len(y))):
            if x[i] == y[j]:
                c[i, j] = 1 + c[i + 1, j + 1]
            else:
                c[i, j] = max(c[i + 1, j], c[i, j + 1])
    return c[0, 0]
```

Recovering LCS: [material covered in recitation]

- to get LCS, not just its length, store parent pointers (like shortest paths) to remember correct choices for guesses:

```
if x[i] = y[j]:
    c[i, j] = 1 + c[i + 1, j + 1]
    parent[i, j] = (i + 1, j + 1)
else:
    if c[i + 1, j] > c[i, j + 1]:
        c[i, j] = c[i + 1, j]
        parent[i, j] = (i + 1, j)
    else:
        c[i, j] = c[i, j + 1]
        parent[i, j] = (i, j + 1)
```

- . . . and follow them at the end:

```
lcs = [ ]
here = (0,0)
while c[here]:
        if x[i] == y[j]:
            lcs.append(x[i])
        here = parent[here]
```

6.006 Introduction to Algorithms
Spring 2008

Lecture 21: Dynamic Programming III: Text Justification, Parenthesization, Knapsack, Pseudopolynomial Time, Tetris Training

Lecture Overview

- Text Justification
- Parenthesization
- Knapsack
- Pseudopolynomial Time
- Tetris Training

Readings

CLRS 15

Review:

* DP is all about subproblems & guessing
* 5 easy steps:
  (a) define subproblems: count ♯ subprobs.
  (b) guess (part of solution): count ♯ choices
  (c) relate subprob. solutions: compute time/subprob.
  (d) recurse + memoize OR build DP table bottom up:
  time = time/subprob. ×♯ subprobs
  (check subproblems related acyclically)
  (e) check original problem = a subproblem or solvable from DP table ($\Longrightarrow$ extra time)
* for sequences, good subproblems are often prefixes OR suffixes OR substrings

Text Justification:

Split text into "good lines"

- obvious (MS Word/Open Office) algorithm: put as many words fit on first line, repeat
- but this can make *very bad* lines

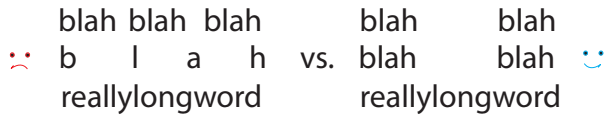


Figure 1: Good vs. Bad Justification

- define badness(i, j) for line of words $[i : j]$ e.g.,

$$\begin{cases} \text{if total length} > \text{page width} \\ (\text{page width - total length})^3 \text{ else} \end{cases}$$

- goal: split words into lines to min $\sum$ badness

1. subproblem = min badness for suffix words$[i :]$
   $\implies$ ♯ subproblems $= \Theta(n)$ where $n =$ ♯ words
2. guessing = where to end first line, say $i : j$
   $\implies$ ♯ choices $= n - i = O(n)$
3. relation:
   - DP$[i] = \min(\text{badness}(i, j) + \text{DP}[j]$ for j in range$(i + 1, n + 1))$
   - DP$[n] = \phi$
     $\implies$ time per subproblem $= O(n)$
4. total time $= O(n^2)$
5. solution = DP$[\phi]$
   (& use parent pointers to recover split)

Parenthesization:

Optimal evaluation of associative expression - e.g., multiplying rectangular matrices

Figure 2: Evaluation of an Expression

2. guessing = outermost multiplication $\underbrace{(\cdots)}_{\uparrow k-1}\underbrace{(\cdots)}_{\uparrow k}$

$\implies \sharp$ choices $= O(n)$

1. subproblems = ~~prefixes & suffixes?~~ NO
   = cost of substring $A[i:j]$

$\implies \sharp$ subproblems $= \Theta(n^2)$

3. Relation:

   - $\text{DP}[i,j] = \min(\text{DP}[i,k] + \text{DP}[k,j] +$ cost of multiplying $(A[i]\cdots A[k-1])$ by $(A[k]\cdots A[j-1])$ for k in range$(i+1,j))$
   - $\text{DP}[i,i+1] = \phi$
     $\implies$ cost per subproblem $= O(n)$

4. total time $= O(n^3)$

5. solution $= DP[0,n]$

(& use parent pointers to recover parens.)

Knapsack:

Knapsack of size S you want to pack

- item i has integer size s_i & real value v_i
- goal: choose subset of items of maximum total value subject to total size $\leq S$

First Attempt:

1. ~~subproblem = value for suffix i:~~ WRONG
2. guessing = whether to include item i $\implies \sharp$ choices $= 2$
3. relation:

- $DP[i] = \max(DP[i+1], v_i + DP[i+1] \text{ if } \cancel{s_i \leq S}?!)$
- not enough information to know whether item i fits - how much space is left? GUESS!

1. subproblem = value for suffix i:
   given knapsack of size X
   $\implies$ ♯ subproblems $= O(nS)$!
3. relation:
   - $DP[i, X] = \max(DP[i+1, X], v_i + DP[i+1, X - s_i] \text{ if } s_i \leq X)$
   - $DP[n, X] = \phi$
     $\implies$ time per subproblem $= O(1)$
4. total time $= O(nS)$
5. solution $= DP[\phi, S]$
   (& use parent pointers to recover subset)
   AMAZING: effectively trying all possible subsets!

Knapsack is in fact NP-complete! $\implies$ suspect no polynomial-time algorithm (polynomial in length of input).

What gives?

- here input $=< S, s_0, \cdots, s_{n-1}, v_0, \cdots, v_{n-1} >$
- length in binary: $O(\lg S + \lg s_0 + \cdots) \approx O(n \lg \ldots)$
- so $O(nS)$ is not "polynomial-time"
- $O(nS)$ still pretty good if S is small
- "pseudopolynomial time": polynomial in length of input & integers in the input

Remember:
polynomial - GOOD
exponential - BAD
pseudopoly - SO SO

Figure 3: Tetris

Tetris Training:

- given sequence of n Tetris pieces & a board of small width w
- must choose orientation & x coordinate for each
- then must drop piece till it hits something
- full rows do not clear
  without these artificialities WE DON'T KNOW! (but: if w large then NP-complete)
- goal: survive i.e., stay within height h

[material below covered in recitation]

First Attempt:

1. ~~subproblem = survive in suffix i:?~~ WRONG
2. guessing = how to drop piece i $\implies$ ♯ choices $= O(w)$
3. ~~relation: $DP[i] = DP[i+1]$~~ ?! not enough information!
   What do we need to know about prefix : i?

   1. subproblem = survive? in suffix i:
   given initial column occupancies $h_0, h_1, \cdots, h_{w-1}$
   $\implies$ ♯ subproblems $= O(n \cdot h^w)$
   3. relation: $DP[i, \boldsymbol{h}] = \max(DP[i, \boldsymbol{m}]$ for valid moves $\boldsymbol{m}$ of piece i in $\boldsymbol{h})$
   $\implies$ time per subproblem $= O(w)$
   4. total time $= O(nwh^w)$
   5. solution $= DP[\phi, \overline{\phi}]$
   (& use parent pointers to recover moves)

6.006 Introduction to Algorithms
Spring 2008

Lecture 22: Dynamic Programming IV: Piano Fingering, Structural DP (Trees), Vertex Cover, Dominating Set, Beyond

Lecture Overview

- Piano Fingering
- Structural DP (trees)
- Vertex Cover & Dominating Set
- Beyond: treewidth, planar graphs, folding

Readings

CLRS 15

Review:

5 easy steps for DP

1. subproblems (define & count)
2. guessing (what & count)
3. relation (the true test)
4. DP (put pieces together)
5. original problem

* 2 kinds of guessing:

A. in 3, guess which other subproblems to use (used by every DP except Fibonacci)

B. in 1, create more subproblems to guess more structure of solution (used by knapsack DP)

- effectively report many solutions to subproblems.
- lets parent subproblem know features of solution.

Piano fingering:

[Parncutt, Sloboda, Clarke, Raekallio, Desain, 1997]
[Hart, Bosch, Tsai 2000]
[Al Kasimi, Nichols, Raphael 2007] etc.

- given musical piece to play, say sequence of (single) notes with right hand
- metric $d(f, p, g, q)$ of difficulty going from note p with finger f to note q with finger g

  e.g., $1 < f < g$ & $p > q \implies$ uncomfortable
  stretch rule: $p \ll q \implies$ uncomfortable
  legato (smooth) $\implies \infty$ if $f = g$
  weak-finger rule: prefer to avoid $g \,\epsilon\, \{4, 5\}$
  $3 \to 4$ & $4 \to 3$ annoying $\sim$ etc.

First Attempt:

1. ~~subproblem = min. difficulty for suffix notes$[i:]$~~
2. ~~guessing = finger f for first note$[i]$~~
3. ~~DP$[i]$ = min$(DP[i+1] + d($note$[i], f,$ note$[i+1],$?$)$ for $f \cdots)$~~
   $\to$ not enough information

1. subproblem = min difficulty for suffix notes[i:] given finger f on first note$[i]$
2. guessing = finger g for next note$[i+1]$
3. DP[inf] = min$(DP[i+1, g] + d($note$[i], f,$ note$[i+1], g)$ for $g \,\epsilon\,$ range$(F))$
   $\leftarrow \sharp$ fingers = 5 for humans
   DP$[n, f] = \phi$
4. Fn subproblems, F choices per subproblem $\implies O(F^2 n)$ time
5. min(DP$[\phi, f]$ for f in range(F))

Structural DP:

Follow combinatorial structure other than a (few)sequence(s) (by analogy to structural vs. regular induction)

* for DP on trees, useful subproblem is subtree rooted at vertex v, for all v

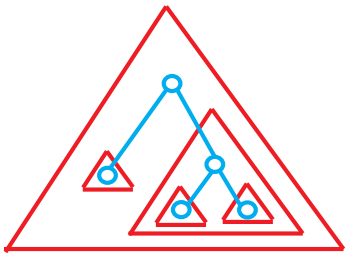


Figure 1: DP on Trees

Vertex Cover:

Find minimum set of vertices (cover) such that every edge is covered on ≥ 1 end

- NP-complete in general graphs
- polynomial for trees:
  1. subproblem = min. cover for subtree rooted at v
     $\implies$ n subproblems
  2. guessing = is v in cover?

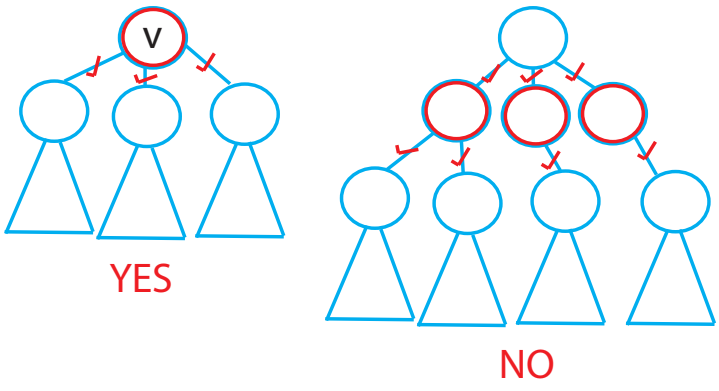


Figure 2: Vertex Cover

- $\implies$ 2 choices
- YES $\implies$ cover children edges
  $\implies$ left with children subtrees
- NO $\implies$ all children must be in cover
  $\implies$ left with grandchildren subtrees

3. DP$[v]$ = min(1 + sum(DP$[c]$ for c in children$[v]$) YES
   len(children) + sum(DP$[g]$ for g in grandchildren(v))) NO
4. time = $O(n)$
5. DP[root]

Dominating set:

Find minimum set of vertices such that every vertex is in or adjacent to set
- again NP-complete in graphs, polynomial on trees.

[material below covered in recitation]

1. subproblem = min. dom. for subtree rooted at v
2. guessing = is v in dom. set?
   - YES $\implies$ dominate children
   - NO $\implies$ must put some child in dom. set
     $\implies$ dominate that child's children
3. DP$[v]$ = min(1 + sum($\underbrace{DP'[c]}$ for c in children$[v]$ *YES*
   but c is already dominated $\cdots$ diff. subprob
   1 + sum($DP(c)$ for $c \neq d$ in children$[v]$)) NO
   + sum($DP'[g]$ for g in children$[d]$)) NO
   again already dominated $\sim$ different subprob
   - guessing of the second type (B)
   for d in children$[c]$)) $\leftarrow$ guess child ϵ set A

1'. subproblem ' = min. dom. for subtree rooted at v given that v dominated already (by parent subproblem)
$\implies$ $2n$ subproblems total

3'. DP'$[v]$ = min(1 + sum($DP'[c]$ for c in children$[v]$, YES
sum($DP[c]$ for c in children$[v]$)) NO

4. time = $O(\sum deg(v)) = O(E) = O(n)$
5. DP[root]

Beyond:

Treewidth:

Many graphs are "thick trees" with reasonable "thickness" (~ 7 e.g.).

- Most problems that are NP-complete in general can be solved in such graphs via DP

Planar Graphs:

Graphs often noncrossing in plane

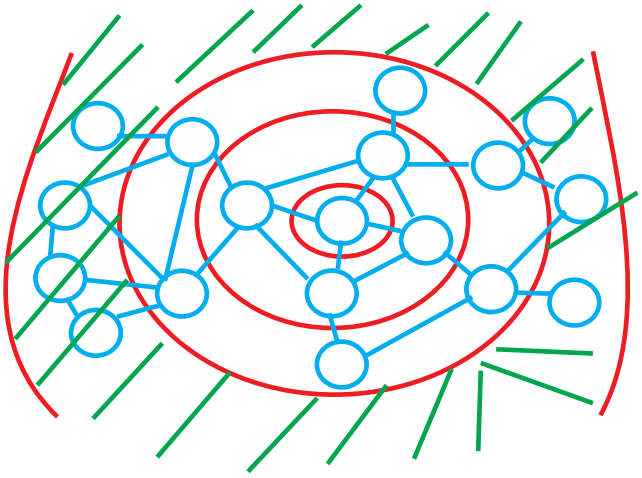


Figure 3: Planar Graphs

- divide planar graph into BFS levels: see Figure 3
- throw away every kth level (e.g., k = 3)
  starting from levels $\phi, 1, \cdots, k-1$ (guess)
- in all cases, remaining graph is a "thick tree" of thickness $O(k)$
  $\Longrightarrow$ can solve this subproblem in poly-time
- can combine these solutions to solve original problem not optimally, but within $1+1/k$ factor of optimal . . . $\forall$ constants k

Folding polygons into polyhedra:

[Metamorphosis of the Cube video]

- DP on substrings of cyclic sequence (polygon)

6.006 Introduction to Algorithms
Spring 2008

Lecture 23: Numerics I

Lecture Overview

- Irrationals
- Newton's Method $(\sqrt{(a)}, 1/b)$
- High precision multiply ←
- Next time
  - High precision radix conversion (printing)
  - High precision division

Irrationals:

Pythagoras discovered that a square's diagonal and its side are incommensurable, i.e., could not be expressed as a ratio - he called the ratio "speechless"!

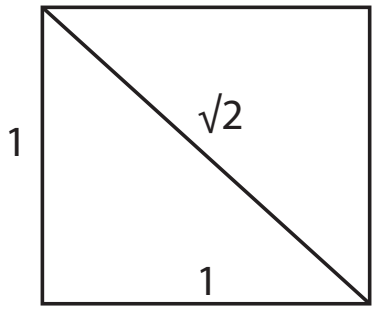


Figure 1: Ratio of a Square's Diagonal to its Sides

Pythagoras worshipped numbers
"All is number"
Irrationals were a threat!

Motivating Question: Are there hidden patterns in irrationals? Can you see a pattern?

$$\sqrt{2} = 1.\ 414\ 213\ 562\ 373\ 095$$
$$048\ 801\ 688\ 724\ 209$$
$$698\ 078\ 569\ 671\ 875$$

Digression

Catalan numbers:
Set P of balanced parentheses strings are recursively defined as

- $\lambda \epsilon P$ (λ is empty string)
- If $\alpha, \beta \epsilon P$, then $(\alpha)\beta \epsilon P$

Every nonempty balanced paren string can be obtained via Rule 2 from a unique α, β pair.
For example, (()) ()() obtained by $\underbrace{(())}_{\alpha}\ \underbrace{()()}_{\beta}$

Enumeration

C_n: number of balanced parentheses strings with exactly n pairs of parentheses
$C_0 = 1$ empty string

C_{n+1}? Every string with $n+1$ pairs of parentheses can be obtained in a unique way via rule 2.

One paren pair comes explicitly from the rule.
k pairs from α, $n-k$ pairs from β

$$C_{n+1} = \sum_{k=0}^{n} C_k \cdot C_{n-k} \quad n \geq 0$$

$$C_0 = 1 \quad C_1 = {C_0}^2 = 1 \quad C_2 = C_0C_1 + C_1C_0 = 2 \quad C_3 = \cdots = 5$$

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796,
58786, 208012, 742900, 2674440, 9694845,
35357670, 129644790, 477638700, 1767263190,
6564120420, 24466267020, 91482563640,
343059613650, 1289904147324, 4861946401452, ...

Geometry Problem

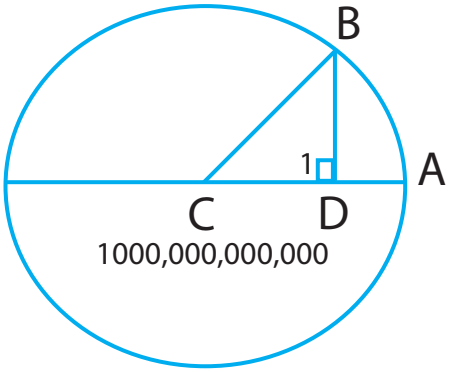


Figure 2: Geometry Problem

$BD = 1$
What is AD?

$$AD = AC - CD = 500,000,000,000 - \underbrace{\sqrt{500,000,000,000^2 - 1}}_{a}$$

Let's calculate AD to a million places!

Newton's Method

Find root of $f(x) = 0$ through successive approximation e.g., $f(x) = x^2 - a$

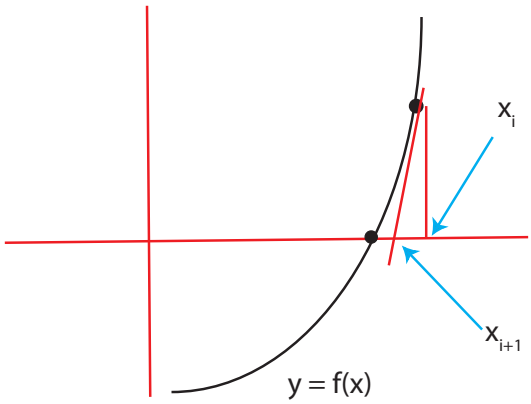


Figure 3: Newton's Method

Tangent at $(x_i, f(x_i))$ is line $y = f(x_i) + f'(x_i) \cdot (x - x_i)$ where $f'(x_i)$ is the derivative.
x_{i+1} = intercept on x-axis

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}}$$

Square Roots

$$f(x) = x^2 - a$$

$$\chi_{i+1} = \chi_i - \frac{(\chi_i{}^2 - a)}{2\chi_i} = \frac{\chi_i + \frac{a}{\chi_i}}{2}$$

Example

$$\begin{aligned} \chi_0 &= 1.000000000 \qquad a = 2 \\ \chi_1 &= 1.500000000 \\ \chi_1 &= 1.416666666 \\ \chi_1 &= 1.414215686 \\ \chi_1 &= 1.414213562 \end{aligned}$$

Quadratic convergence, ♯ digits doubles

High Precision Computation

$\sqrt{2}$ to d-digit precision: $\underbrace{1.414213562373}_{\text{d digits}}\cdots$
Want integer $\lfloor 10^d\sqrt{2} \rfloor = \lfloor \sqrt{2 \cdot 10^{2d}} \rfloor$ - integral part of square root
Can still use Newton's Method.
Let's try it on $\sqrt{2}$, and our segment AD!
See anything interesting?

High Precision Multiplication

Multiplying two n-digit numbers (radix $r = 2, 10$)
$0 \le x, y < r^n$

$$\begin{aligned} x &= x_1 \cdot r^{n/2} + x_0 \qquad x_1 = \text{high half} \\ y &= y_1 \cdot r^{n/2} + y_0 \qquad x_0 = \text{low half} \\ 0 &\le x_0, x_1 < r^{n/2} \\ 0 &\le y_0, y_1 < r^{n/2} \end{aligned}$$

$$z = x \cdot y = x_1 y_1 \cdot r^n + (x_0 \cdot y_1 + x_1 \cdot y_0) r^{n/2} + x_0 \cdot y_0$$

4 multiplications of half-sized ♯'s $\implies$ quadratic algorithm $\theta(n^2)$ time

Karatsuba's Method

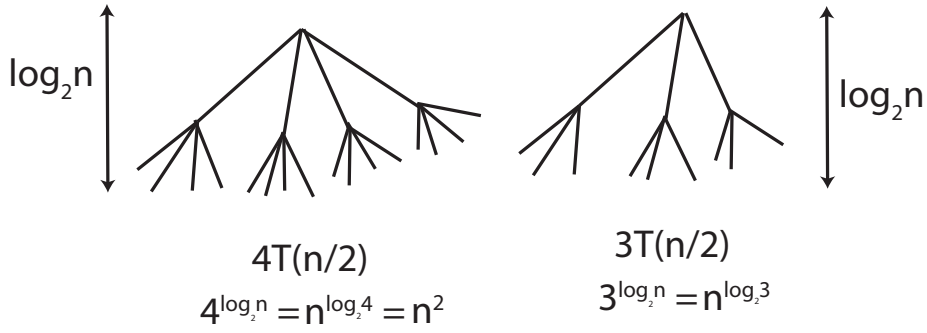


Figure 4: Branching Factors

Let

$$\begin{aligned} z_0 &= \underline{x_0 \cdot y_0} \\ z_2 &= x_2 \cdot y_2 \\ z_1 &= (x_0 + x_1) \cdot (y_0 + y_1) - z_0 - z_2 \\ &= x_0 y_1 + x_1 y_0 \\ z &= z_2 \cdot r^n + z \cdot r^{n/2} + z_0 \end{aligned}$$

There are three multiplies in the above calculations.

$$\begin{aligned} T(n) &= \text{time to multiply two } n\text{-digit} \sharp' s \\ &= 3T(n/2) + \theta(n) \\ &= \theta\left(n^{\log_2 3}\right) = \theta\left(n^{1.5849625\cdots}\right) \end{aligned}$$

Better than $\theta(n^2)$. Python does this.

Error Analysis of Newton's Method

Suppose $X_n = \sqrt{a} \cdot (1 + \epsilon_n)$ ϵ_n may be + or -
Then,

$$\begin{aligned} X_{n+1} &= \frac{X_n + a/X_n}{2} \\ &= \frac{\sqrt{a}(1+\epsilon_n) + \frac{a}{\sqrt{a}(1+\epsilon_n)}}{2} \\ &= \sqrt{(a)}\frac{\left((1+\epsilon_n) + \frac{1}{(1+\epsilon_n)}\right)}{2} \\ &= \sqrt{(a)}\left(\frac{2 + 2\epsilon_n + {\epsilon_n}^2}{2(1+\epsilon_n)}\right) \\ &= \sqrt{(a)}\left(1 + \frac{{\epsilon_n}^2}{2(1+\epsilon_n)}\right) \end{aligned}$$

Therefore,

$$\epsilon_{n+1} = \frac{{\epsilon_n}^2}{2(1+\epsilon_n)}$$

Quadratic convergence, as ♯ digits doubles.

6.006 Introduction to Algorithms
Spring 2008

Lecture 24: Numerics II

Lecture Overview

- Review:
  - high precision arithmetic
  - multiplication
- Division
  - Algorithm
  - Error Analysis
- Termination

Review:

Want millionth digit of $\sqrt{2}$:

$$\lfloor \sqrt{2 \cdot 10^{2d}} \rfloor \; d = 10^6$$

Compute $\lfloor \sqrt{a} \rfloor$ via Newton's Method

$$\begin{aligned} \chi_0 &= 1 \quad \text{(initial guess)} \\ \chi_{i+1} &= \frac{\chi_i + a/\chi_i}{2} \quad \leftarrow \text{division!} \end{aligned}$$

Converges quadratically; ♯ correct digits doubles each step.

Multiplication:

1. Naive Divide & Conquer method: $\theta(d^2)$ time
2. Karatsuba: $\theta(d^{\log_2 3}) = \theta(d^{1.584\ldots})$
3. Toom-Cook generalizes Karatsuba (break into $k \geq 2$ parts)

$$T(d) = 5T(d/3) + \theta(d) = \theta\left(d^{\log_3 5}\right) = \theta\left(d^{1.465\ldots}\right)$$

4. Schonhage-Strassen - almost linear! $\theta(d \lg d \lg \lg d)$ using FFT. All of these are in gmpy package
5. Furer(2007): $\theta\left(n \log n 2^{O(\log^* n)}\right)$ where $\log^* n$ is iterated logarithm. ♯ times log needs to be applied to get a number that is less than or equal to 1.

High Precision Division

We want high precision rep of $\frac{a}{b}$

- Compute high-precision rep of $\frac{1}{b}$ first
- High-precision rep of $\frac{1}{b}$ means $\lfloor \frac{R}{b} \rfloor$ where R is large value s.t. it is easy to divide by R
  Ex: $R = 2^k$ for binary representations

Division

Newton's Method for computing $\frac{R}{b}$

$$\begin{aligned} f(x) &= \frac{1}{x} - \frac{b}{R} \quad \left(\text{zero at } x = \frac{R}{b}\right) \\ f'(x) &= \frac{-1}{x^2} \\ \chi_{i+1} &= \chi_i - \frac{f(\chi_i)}{f'(\chi_i)} = \chi_i - \frac{\left(\frac{1}{\chi_i} - \frac{b}{R}\right)}{-1/{\chi_i}^2} \\ \chi_{i+1} &= \chi_i + {\chi_i}^2\left(\frac{1}{\chi_i} - \frac{b}{R}\right) = 2\chi_i - \frac{b{\chi_i}^2 \rightarrow \text{multiply}}{R \rightarrow \text{easy div}} \end{aligned}$$

Example

Want $\frac{R}{b} = \frac{2^{16}}{5} = \frac{65536}{5} = 13107.2$

Try initial guess $\frac{2^{16}}{4} = 2^{14}$

$$\begin{aligned} \chi_0 &= 2^{14} = 16384 \\ \chi_1 &= 2 \cdot (16384) - 5(16384)^2/65536 = \underline{12}288 \\ \chi_2 &= 2 \cdot (12288) - 5(12288)^2/65536 = \underline{13}056 \\ \chi_3 &= 2 \cdot (13056) - 5(13056)^2/65536 = \underline{13107} \end{aligned}$$

Error Analysis

$$\begin{aligned}
\chi_{i+1} &= 2\chi_i - \frac{b{\chi_i}^2}{R} \qquad \text{Assume } \chi_i = \frac{R}{b}(1+\epsilon_i) \\
&= 2\frac{R}{b}(1+\epsilon_i) - \frac{b}{R}\left(\frac{R}{b}\right)^2(1+\epsilon_i)^2 \\
&= \frac{R}{b}\left((2+2\epsilon_i) - (1+2\epsilon_i+{\epsilon_i}^2)\right) \\
&= \frac{R}{b}\left(1-{\epsilon_i}^2\right) = \frac{R}{b}(1+\epsilon_{i+1}) \text{ where } \epsilon_{i+1} = -{\epsilon_i}^2
\end{aligned}$$

Quadratic convergence; ♯ digits doubles at each step
Therefore complexity of division = complexity of multiplication

Termination

Iteration: $\chi_{i+1} = \lfloor \frac{\chi_i + \lfloor a/\chi_i \rfloor}{2} \rfloor$
Do floors hurt? Does program terminate?
Iteration is

$$\begin{aligned}
\chi_{i+1} &= \frac{\chi_i + \frac{a}{\chi_i} - \alpha}{2} - \beta \\
&= \frac{\chi_i + \frac{a}{\chi_i}}{2} - \gamma \quad \text{where } \gamma = \frac{\alpha}{2} + \beta \text{ and } 0 \le \gamma < 1
\end{aligned}$$

Since $\frac{a+b}{2} \ge \sqrt{ab}$, $\frac{\chi_i + \frac{a}{\chi_i}}{2} \ge \sqrt{a}$, so subtracting γ always leaves us $\ge \lfloor \sqrt{a} \rfloor$. This won't stay stuck above if $\epsilon_i < 1$ (good initial guess)

6.006 Introduction to Algorithms
Spring 2008

Lecture 25: Beyond 6.006: Follow-on Classes, Geometric Folding Algorithms

Algorithms Classes at MIT: (post 6.006)

1. 6.046: Intermediate Algorithms (more advanced algorithms & analysis, less coding)
2. 6.047: Computational Biology (genomes, phylogeny, etc.)
3. 6.854: Advanced Algorithms (intense survey of whole field)
4. 6.850: Geometric Computing (working with points, lines, polygons, meshes, . . .)
5. 6.851: Advanced Data Structures (sublogarithmic performance)
6. 6.852: Distributed Algorithms (reaching consensus in a network with faults)
7. 6.855: Network Optimization (optimization in graph: beyond shortest paths)
8. 6.856: Randomized Algorithms (how randomness makes algorithms simpler & faster)
9. 6.857: Network and Computer Security (cryptography)
10. 6.885: Geometric and Folding Algorithms * TODAY

Other Theory Classes:

- 6.045: Automata, Computability, & Complexity
- 6.840: Theory of Computing
- 6.841: Advanced Complexity Theory
- 6.842: Randomness & Computation