Chapter 5: Algorithms and Heuristics

CS105: Great Insights in Computer Science

- problems on lists of numbers
- algorithm vs. heuristic
- heuristic search in subset sum?
Here’s Where We Stand

- Up until now, we discussed how a computer could be created starting from bits and wires and working up to a high-level language.

- In classic CS style (reduction!), we now take all these lower levels for granted and build on them to create new capabilities.

- The next block of lectures takes a high-level language as our starting point. So, it would help for you to hear a bit more details about it.

Python Tutorial

- Although you don’t need to learn to program in this class, I’d like you to be able to read a simple program to see what it does.
**Variables and Strings**

```
print "hello"  # hello
it = "hello"
print it      # hello
print "it"    # it
print 'it' + " " + it  # it hello
x = "tuna"
y = "fish"
print x       # tuna
print x + y   # tunafish
print x + " " + y  # tuna fish
print "x + y"  # x + y
```

**Subroutines**

```
def d(z):
    print z + " are delicious"

d("salmon")  # salmon are delicious
d(x)           # tuna are delicious
d(y)           # fish are delicious
d(x+y)         # tunafish are delicious
d("x+y")      # x+y are delicious
```
Functions

```python
def s(w):
    return "fried " + w

print s("potatoes")  # fried potatoes
print s(x)  # fried tuna
d(s("eggs"))  # fried eggs are delicious
print s(x) + s(y)  # fried tuna fried fish
```

Conditional Branch

```python
num = 17
if num > 10:
    print "multidigit"  # multidigit
if num % 2 == 0:
    print "even"
else:
    print "odd"  # odd
if num < 10:
    print "single digit"
```
Lists

```
z = ["Paul", "George",
    "Ringo", John"]

print z  # ['Paul', 'George', 'Ringo', 'John']
print z[0]  # Paul
print z[3]  # John
print z[1:]  # ['George', 'Ringo', 'John']
print z + ["Stuart", "Billy"]  # ['Paul', 'George', 'Ringo', 'John', 'Stuart', 'Billy']
p
print len(z)  # 4
print range(4)  # 0, 1, 2, 3
```

Strings as Lists

```
x = tuna
print x  # tuna
print x[0]  # t
print x[1:]  # una
print (s(y))[1:8]  # ried fi
def reverse(s):
   if s == "": return ""
   return reverse(s[1:]) + x[0]
reverse("swordfish")  # hsifdrows
del z[2]
print z  # ['John', 'Paul', 'Ringo']
```
Numbers

print 1+1, 2-2, 3*3, 4/4, 10/3 2 0 9 1 3
print 1 + x <error>
print str(1) + x 1tuna
def frac(x,y):
    print x/y, x-y*(x/y), "/", y
frac(1,3) 0 1 / 3
frac(14,3) 4 2 / 3

Loops

for b in z:
    print b + " was a Beatle."

Paul was a Beatle.
George was a Beatle.
Ringo was a Beatle.
John was a Beatle.
x = 1000; y = 1
while x > 1:
    y = y + 1; x = x / 2
print y 10
Next Goal

- We looked at different ways of writing programs to produce the same output (Macdonald #1, #2, and #3, for example).
- None was definitively better, except aesthetically.
- We’ll look at another way of comparing programs...

Sock Matching

- Hillis begins Chapter 5 with an example.
- We’ve got a basketful of mixed up pairs of socks.
- We want to pair them up reaching into the basket as few times as we can.
Sock ‘Ops

- **getSock()**: pulls a sock out of the basket and provides its value.
- **match(sock1, sock2)**: takes two socks and returns True if they match (and pairs them) and False otherwise.
- **replaceSock(sock)**: puts the given sock back in the laundry basket.
- **emptyBasket()**: returns True if the basket is empty and False if there are still more socks.

Sock Sorter A

- Grab two socks.
- If they don’t match, toss them back in the basket.
- Will this procedure ever work?
- Will it *always* work?

```python
def sockA():
    x = getSock()
    y = getSock()
    if not match(x,y):
        replaceSock(x)
        replaceSock(y)
```
Measuring Performance

• Hillis asserts that the time-consuming part of this operation is reaching into the basket: getSock().

• Let’s say we have 8 pairs of socks.

• How many getSock() operations are done by sockA() do?

• Min?
• Max?
• Average?

• How do these values grow with increasing numbers of pairs of socks?

Sock Sorter B

• Make a pile.
• Grab a sock (as long as there is 1).
• Look for its mate in the pile.
• If found, shrink pile.
• If not, add to the pile.
• Min/Max/Mean?

```python
def sockB():
    pile = []
    while not emptyBasket():
        x = getSock()
        matched = False
        for i in range(len(pile)):
            if not matched and match(x, pile[i]):
                matched = True
                del pile[i]
        if not matched:
            pile = pile + [x]
```
Analysis

• Gets every sock exactly once!

• A bit of extra work keeping the pile in proper shape.

• How might this approach be considered less good than the previous approach?

Repeat For Each Sock

sockA()

• Do you have a matching pair? Set it aside.

• Do you have a non-matching paper? Put them back in the basket.

sockB()

• Is there a match on the table? Pair them and set the pair aside.

• Otherwise, find an empty place on the table and set the sock down.
Notable if No Table

- One disadvantage of sockB() is that you must have a big empty space available.
- What if you can only hold two socks at a time?
- Suggest for a competitor for sockA()?

Sock Sorter C

- Grab two socks.
- If they don't match, put one back and grab a replacement.
- Repeat until a match is found.
- Ever? Always? Min, max, average? Better/worse/same?

```python
def sockC():
    x = getSock()
    y = getSock()
    while not match(x,y):
        replaceSock(y)
        y = getSock()
```
Round #2

sockA()

• Do you have a matching pair? Set it aside.

• Do you have a non-matching paper? Put them both back in the basket.

sockC()

• Do you have a matching pair? Set it aside.

• Do you have a non-matching paper? Put one back in the basket.

Analysis of sockC()

• Roughly the same number of matching operations, but since it always holds onto one sock, roughly half the number of getSocks().

• When might this approach fail in the real world?

• Does sockA() suffer from this difficulty?
Algorithms

- sockA(), sockB(), and sockC() represent different approaches to solving the problem of sock sorting.
- A concrete approach to solving a problem is called an algorithm.
- Different algorithms can be better or worse in different ways.

Lessons Learned

- If we have a notion of “time” (getSock() or number of statements executed), we can compare different algorithms based on the time they take.
- They really are different, so use good algorithms.
- I once redesigned a colleague’s algorithm and it ran in seconds where it used to take an hour.
- Hard to believe they solved the same problem.
You Think You Have Problems

- The sock matching problem is a little silly, because the right answer seems sort of obvious.

- Algorithms researchers pride themselves on finding simple-sounding problems for which the best algorithm is not obvious.

- Ah, but what’s a problem? It should have a well-defined input and a well-defined output.

Decision Problems on Lists

- A natural class of problems is decision problems on lists of numbers. That is, the input is a list of numbers and the output is a decision, essentially a bit (True or False, yes or no).

- We’ll try some examples and see if we can find efficient algorithms for solving different problems.
Finding the Max

- Problem: “Is the max x”? Given a list of numbers and a number x, is the largest number in the list equal to x?
- Is the max 98?

48, 40, 14, 46, 31, 0, 27, 12, 22, 71, 45, 63, 30, 64, 83, 28, 97, 90, 85, 52

How Do You Do It?

- Specifying an algorithm to the computer is like talking to a little kid. You need to spell out everything precisely.

- Let’s try.

```python
def isMax(x):
    global l
    largest = 0
    for i in list:
        if l[i] > largest:
            largest = l[i]
    if x == largest: return True
    return False
```
Finding the Median

- Problem: “Is the median $x$”? Given a list of numbers and a number $x$, is the median number in the list equal to $x$?

- Is the median 45?

48, 40, 14, 46, 31, 0, 27, 12, 22, 71, 45, 63, 30, 64, 83, 28, 90, 85, 52

Sum Divisible By 5?

- Problem: “Is the sum divisible by 5”? Given a list of numbers, is the sum of the numbers divisible by 5?

- Is the sum divisible by 5?

48, 40, 14, 46, 31, 0, 27, 12, 22, 71, 45, 63, 30, 64, 83, 28, 90, 85, 52
Product Divisible By 5?

• Problem: “Is the product divisible by 5”? Given a list of numbers, is the product of the numbers divisible by 5?

• Is the product divisible by 5?

48, 41, 14, 46, 31, 2, 27, 12, 22, 71, 44, 63, 33, 64, 83, 28, 96, 87, 52

How Do It Fast? Median

• If there are 19 numbers, median is the one in the 10th position when the list is sorted.

• Straightforward algorithm is: (1) sort the list, (2) find the median in the sorted list, (3) check if it equals \( x \).

• That’s fine, but, there’s a shortcut: (1) count the number of numbers less than \( x \) (\( L \)) and the number equal to \( x \) (\( E \)), (2) check if \( L < 10 \) and \( L+E > 9 \).

• Let’s try it! Is the median 47?

9, 13, 2, 24, 13, 60, 31, 62, 59, 70, 47, 47, 77, 46, 46, 70, 39, 6, 63
How Do It Fast? Sum

• Straightforward algorithm is: (1) sum up the numbers in the list, (2) divide the grand total by 5, (3) check if there’s no remainder.

• There are a few shortcuts we can use: (A) A number is divisible by 5 if and only if it ends in 5 or 0. (B) The sum of two numbers is divisible by 5 only if the sum of their last digits is divisible by 5.

• Faster algorithm: (1) Keep a running total of the last digits, keeping only the last digit of the sum, (2) check if it’s 0 or 5. Let’s try: Is the sum divisible by 5?

9, 13, 2, 24, 13, 60, 31, 62, 59, 70, 47, 47, 77, 46, 46, 70, 39, 6, 63

How Do It Fast? Product

• Straightforward algorithm is: (1) multiply all the numbers in the list, (2) divide the product by 5, (3) check if there’s no remainder.

• There are another shortcut we can use: A product is divisible by 5 if and only if at least one of the multiplicands is divisible by 5.

• Faster algorithm: Check if any number in the list ends with 0 or 5. Let’s try. Is the product divisible by 5?

9, 13, 2, 24, 13, 61, 31, 62, 59, 72, 47, 45, 77, 46, 46, 73, 39, 6, 63
def isSum5():
    total = 0
    for i in list:
        total = total + (l[i] % 10)
        total = total % 10
    if total == 0 or total == 5:
        return True
    return False

def isProd5():
    for i in list:
        if l[i] % 10 == 0:
            return True
        if l[i] % 10 == 5:
            return True
    return False

Comparing Algorithms

Next, we’ll talk about how computer scientists compare algorithms to decide which is better.

Could try them all, but that would defeat the purpose, wouldn’t it?

But first, an aside to introduce some mathematical concepts.
Let’s talk about how many syllables we sing given a song of a certain type as the number of verses grows.

In general, we’re interested in the number of syllables as a function of $n$, the number of verses.

Generalized Dreidel Song

1. I had a little dreidel, I made it out of clay And when it’s dry and ready Oh dreidel I shall play.

Chorus:
Oh dreidel dreidel dreidel I made it out of clay And when it’s dry and ready Oh dreidel I shall play.

2. I had a little dreidel, I made it out of plastic If someone steals my dreidel I’ll do something very drastic.

Chorus

3. I had a little dreidel, I made it out of glass My mom said when I spin it, to spin it on the grass.

Chorus

4. I had a little dreidel, I made it out of chocolate, but when I went to spin it, it had melted in my pocket.

Chorus

5. I had a little dreidel, I made it out of wood, and when I went to spin it, it spun just like it should.

Chorus

6. I had a little dreidel, I made it out of ice, but when I went to spin it, it melted...that’s not nice!!

Chorus

7. I had a little dreidel, I made it out of mud, and when I went to spin it, it fell down with a thud.

Chorus

8. I had a little dreidel, I made it out of tin, I made it kind of crooked, and so I always win.

Chorus
## Counting Syllables

<table>
<thead>
<tr>
<th>verses</th>
<th>syllables</th>
<th>verses</th>
<th>syllables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51</td>
<td>5</td>
<td>271</td>
</tr>
<tr>
<td>2</td>
<td>109</td>
<td>6</td>
<td>323</td>
</tr>
<tr>
<td>3</td>
<td>161</td>
<td>7</td>
<td>375</td>
</tr>
<tr>
<td>4</td>
<td>219</td>
<td>8</td>
<td>432</td>
</tr>
</tbody>
</table>

## Plotting Syllables

- Total syllables roughly, $T(n) = 54 \cdot n$. 

![Plot of syllables vs verses](image-url)
Old Macdonald: Verse 4

Old Macdonald had a farm, E-I-E-I-O
And on his farm he had a chick, E-I-E-I-O
With a "cluck, cluck" here and a "cluck, cluck" there
Here a "cluck" there a "cluck"
Everywhere a "cluck-cluck"
With a "neigh, neigh" here and a "neigh, neigh" there
Here a "neigh" there a "neigh"
Everywhere a "neigh-neigh"
With a (snort) here and a (snort) there
Here a (snort) there a (snort)
Everywhere a (snort-snort)
With a "moo-moo" here and a "moo-moo" there
Here a "moo" there a "moo"
Everywhere a "moo-moo"
Old Macdonald had a farm, E-I-E-I-O

25 syllables
22 syllables
22 syllables
22 syllables
12 syllables
37 + 4 x 22
= 125 syllables

4-verse song: (37 + 1 x 22) + (37 + 2 x 22)
+ (37 + 3 x 22) + (37 + 4 x 22) = 368 syllables

Counting Syllables

<table>
<thead>
<tr>
<th>verses</th>
<th>syllables</th>
<th>verses</th>
<th>syllables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59</td>
<td>5</td>
<td>515</td>
</tr>
<tr>
<td>2</td>
<td>140</td>
<td>6</td>
<td>684</td>
</tr>
<tr>
<td>3</td>
<td>243</td>
<td>7</td>
<td>875</td>
</tr>
<tr>
<td>4</td>
<td>368</td>
<td>8</td>
<td>1088</td>
</tr>
</tbody>
</table>
• Total syllables?
Summing Syllables

- Verse $i$ has $37 + 22 \times i$ syllables.
- Song with $n$ verses:
  \[
  (37 + 1 \times 22) + (37 + 2 \times 22) + \ldots + (37 + n \times 22)
  \]
  \[
  = 37n + (1 + 2 + \ldots + n) \times 22
  \]

Sum of $n$ Integers

- Old McDonald with $n$ verses:
  \[
  37n + 22 \times (1 + 2 + \ldots + n) = 11n^2 + 48n
  \]
**N Bottles of Beer**

- **Verse 99:**
  - 99 bottles of beer on the wall.
  - 99 bottles of beer.
  - If one of those bottles should happen to fall.
  - 98 bottles of beer on the wall.

- **Verse i:**
  - 29 syllables + 2 x syllables in i + syllables in i-1.

- **Syllables in i?**
  - Roughly the number of digits in i.
  - Very slow growing function... by googol, only reaches 101.

**Logarithms**

- **lg 100 = 2**
- **lg 10000 = 4**
- **lg x:** roughly the number of times you can divide x by 10 before you reach 1 or less.

- **Other logs:**
  - ln is the natural logarithm (base e)
  - log is base 2 logarithm: number of times you can halve before reaching 1 or less.
### Whole Song

- So, syllables in verse \(i\) of \(N\) Bottles of Beer:
  - Approximately, \(29 + 3 \lg i\).

- \(n\) verses: \(29n + 3 (\lg 1 + \lg 2 + \cdots + \lg n)\)
  - 90% of sum is \(\lg n\), 9% is \(\lg n - 1\), 0.9% is \(\lg n - 2\), 0.09% is \(\lg n - 2\), ...
  - Approximately, \(2.8 (n \lg n) + 29n\).
Whole Song

- $n$ verses:
  
  $$(11 + 4 + \lg 1) + (11 + 8 + 2 \lg 2) +$$
  
  $$(11 + 12 + 3 \lg 3) + \ldots + (11 + 4n + n \lg n)$$
  
- Approx., $11n + 2n(n+1) + .46n(n+1) \lg n$.
  
- Approx., $.46n^2 \lg n + 2n^2 + .46n \lg n + 13n$.

Different Growth Rates

- With these constants, $n^2 \lg n$ has fastest growth, then quadratic, then $n \lg n$, then linear.
  
  | $0.4n^2$ (Quadratic) |
  | $2.86n \lg n$ |
  | $0.572n^2 \lg n$ |

- For big $n$, always the same order regardless of the constants!

- Leads to the notion of “Big O”.
Big O

- Formally, big O is a notation that denotes a class of functions all of which are upper bounded asymptotically.

- In practice, however, it gives us a way of ignoring constants and low-order terms to cluster together functions that behave similarly.

\[ O(n) \]

Common Growth Classes

- Linear: \( O(n) \)
  - Dreidel
  - Clementine
- Quadratic: \( O(n^2) \)
  - An Old Lady
  - Old Macdonald
  - There Was a Tree
- \( O(n \log n) \)
  - \( N \) Bottles of Beer
  - \( N \) Little Monkeys
- \( O(n^2 \log n) \)
  - \( N \) Days of Christmas
  - Who Knows \( N \)?

- Each verse contains the next higher number.
- Each verse lists one more number than the previous.
- Each verse a constant size larger than the previous.
- Constant size verse.
Another Visualization

- Linear (O(n))
- Quadratic (O(n^2))
- O(n log n)

Non-Classical Songs

- As far as I know, classical songs are all linear (O(n)), quadratic (O(n^2)), O(n log n), and O(n^2 log n). [Extra credit for discovering another type!]
- Nevertheless, I can make up a few more songs to demonstrate a few other important growth rates.
Skip A Few...

- My kids used to play this game: “I can count up to 100. One, two, skip-a-few, 99, 100!”.
  Or “One, two, skip-a-few, 999, 1000!”.
- Number of syllables to “skip count” to $n$?
  - $5 + 2 \log n$: This song is $O(\log n)$.
- With exponential notation: “One, two, skip-a-few, $10^{100}-1$, $10^{100}$. Now, the syllables depend on the number of digits: $O(\log \log n)$.

Very Slow Song

- On the flip side, consider a song in which verse $i$ consists of singing all the numbers with exactly $i$ digits.
- Now, a song with $n$ verses is $O(10^n)$.
- This is an exponential growth. Something I’d like to say a bit more about.
Exponential Growth

• ipods.
• Computer speed: Moore’s Law.
• World Population.
• Bacterial growth (while the food lasts).
• Spam.

Pet Peeve Alert

• Because exponential growth rates are so common, the phrase has entered the public lexicon.
• Not always properly... Many people seem to use it to mean “a lot more”, which doesn’t really make sense.
• Let’s learn to recognize the proper use, ok?
Which Are Correct?

• Source: Newsweek.
• The country desperately needs to upgrade its roads and seaports, and to **exponentially** increase agricultural and manufactured exports.
  
  \[
  \text{exports}(t) = 10^t
  \]
• Exponentially less expensive than a 20-hour flight to the Bushveld of South Africa or the remote rain forests of Costa Rica, domestic safaris can be nearly as exciting—and far more accessible for families with kids.

Continued

• Demand for IVF treatments, which climbed **exponentially** during the past 20 years, has plateaued.
  
  \[
  \text{demand}(t) = 10^t
  \]
• Consequently, an unintended but **exponentially** growing number of middle-class Americans is being affected.
  
  \[
  \text{affectedpeople}(t) = 10^t
  \]
• I have been on television for almost 12 years, and in that relatively short time I’ve seen the medium change **exponentially**.

• Now in the tsunami’s aftermath, global health experts worry that the dangerous microbes already lurking in underdeveloped regions of Asia will spread **exponentially**, pushing the tsunami’s enormous death toll even higher.
  
  \[
  \text{affectedArea}(t) = 10^t
  \]
• Injury rates [for cheerleaders] are **exponentially** higher for a flier than for a footballer," says NCCSI's Robert Cantu.
Algorithm Analysis

- Now, that we have a sense of how various quantities grow as a function of other quantities.
- Let’s apply this idea to analyzing our sock sorters.
- For each algorithm, how does the number of calls to getSock() grow as a function of the number of pairs of socks $n$?

Random Probability Facts

- Since sock sorting setting involves probabilities, it helps to review a few facts.
- If an event happens on each try with probability $p$, we’d expect $1/p$ tries (on average) before we’re successful. Example: Average number of die rolls before getting a 3 (probability 1/6) is 6.
- If we look through a randomly ordered list of length $n$ for an item on the list, on average we’ll need to look through $n/2$ of the list.
Analyzing Sock Sorting

• How many calls to 
getSocks does sockA() 
take to sort 50 pairs of 
socks?

• sockA(): choose a 
random pair. Return 
to basket if no match.

• Number of getSocks 
before a pair is found?

• Probability of a match 
is 1/99.

• Number of tries before 
match found? 99, on 
average.

• Each of the 99 tries 
calls getSock twice, so 
198 for the first pair, 
on average.

sockA(), Continued

• So, how many calls to 
getSock() to find the 
first pair given \( n \) pairs 
in the basket? \( 2(2n-1) \) 
\[= 4n-2. \]

• Now, there are \( n-1 \) 
pairs left. Finding the 
second pair will take 
\( 4(n-1)-2 = 4n-6 \) calls.

• When there is one pair 
left, it takes 2 calls.

• Total 
\[= 2 + 6 + 10 + \ldots + 4n-2 \]
\[= 4(1+2+\ldots+n)-2n \]
\[= 4 \frac{n(n+1)}{2} - 2n \]
\[= 2n^2. \]

• So, \( O(n^2) \) algorithm.
Intuitive Analysis

• Since the time to find each pair is proportional to the number of pairs left, the total amount of time until all pairs are found is order $n^2$.

• sockC() is the same, except the time is halved. Still order $n^2$.

How about sockB()?

• sockB(): Keep a pile on the table. Grab a sock and check if its mate is on the bed. If not, add it to the pile.

• Since all socks are matched up and no socks are returned to the basket, there is precisely one getSock() call per sock, $2n$ if $n$ pairs.

• So, a $O(n)$ algorithm! Linear, order $n$, etc.

• No wonder it’s fast.
Algorithm Design Goal

• Not just trying to solve a problem, but solve it well with respect to some goal.

• Best way to the airport?
  - Time?
  - Money?
  - Gas?
  - What else?

Another Decision Problem

• Problem: “Is there a pair that sums to \( x \)”? Given a list of numbers, is there a pair of distinct numbers in the list that sums up to \( x \)?

• Is there a pair that sums to 86?

48, 41, 14, 46, 31, 2, 27, 12, 22, 71, 44, 63, 33, 64, 83, 28, 96, 87, 52

• What is the order of the runtime of your approach?
Yet Another Problem

- Problem: “Is there a subset that sums to $x$”? Given a list of numbers, is there a subset of distinct numbers in the list that sums up to $x$?

- Is there a subset that sums to 433?

48, 41, 14, 46, 31, 2, 27, 12, 22, 71, 44, 63, 33, 64, 83, 28, 96, 87, 52

- Does this problem seem harder? Is it solvable?

NP-complete Problems

- There’s another remarkable class of problems:
  - They *can* be solved (not incomputable).
  - The best algorithms we have take *exponential* time.
  - They are *ubiquitous*.
  - No one has proven they *can’t* be solved efficiently.
  - If we could solve *one* of them efficiently, we could solve *all* of them efficiently. (Reduction again!)
NP and Puzzles

- Most puzzles can be thought of as NP problems. Why? Because the answer appears the next day. It might be hard to find the answer, but it’s easy to check once you hear it.
- Sometimes I say NP stands for “Nice Puzzle” for this reason.

Graph Algorithms

- Next, we’re going to look at several critical ideas in algorithm design and use the example of Google to motivate them.
- Our question: How does Google find stuff?
The Internet (a piece)

A Conversation

Hi, I'm porthos.rutgers.edu.

To: porthos.rutgers.edu
What gives?

To: dir.yahoo.com
I'm told you have a web page called "Science/Computer_Science/College_and_University_Departments/?b=20". Can you send me a copy?

To: porthos.rutgers.edu
Sure:
Web Search

- All web browsing consists of requests for specific pages.
- But, what happens if we don’t know what we want?
- A “search engine” is just a web server that can respond to a particular request for web pages.
Another Conversation

Porthos asked Google where it could find pages about “rutgers computer science”.

Google responded with a page that included addresses of other pages.

Porthos can now request those pages directly from the web servers that “host” (store and dispense) them.

Indirection
How Does Google Know?

- So, somehow, Google has to put together a web page in response to any query, which includes a list of names of pages that contain those terms.

- But, how does it know which pages contain which terms?

Theories?
1.
2.
3.

An Experiment

http://www.cs.rutgers.edu/~mlittman/courses/cs442-06/googletest1.html

How Does Google Find Pages?

This page has little purpose other than to include the word "googlediscovery". This is a word (sort of) that I concocted on January 3, 2006 and verified that it was unknown to Google. (Did you mean: "google discovery"). This page has a direct link from the course homepage. Sure, it's no "truthiness", but it's still useful scientifically.

There is a secondary page, http://www.cs.rutgers.edu/~mlittman/courses/cs442-06/googletest2.html, that does not have an explicit link from anywhere. It has its own special term, which consists of concatenating "google" and "blackout".

I also made a third term, formed from "google" and "abyss", that I do not plan on putting on any page. (Did you mean: google's)

- 1/03/06 (8:01am): Saved this page.
- 1/03/06 (8:03am): "discovery" (0), "blackout" (0), "abyss" (0).
- 1/04/06 (8:00am): "discovery" (0), "blackout" (0), "abyss" (0).
- 1/05/06 (8:10am): "discovery" (0), "blackout" (0), "abyss" (0).
- 1/06/06 (9:18am): "discovery" (0), "blackout" (0), "abyss" (0).
- 1/07/06 (8:39am): "discovery" (0), "blackout" (0), "abyss" (0).
- 1/08/06 (8:00am): "discovery" (1), "blackout" (0), "abyss" (0).
What Do You Think Now?

• Google knew the word “googlediscovery” five days after I put up a web page with the word and linked it to the course web page.

• Google still didn’t know the word “googleblackout” more than a month later in spite of being on a similar (but unlinked) page at the same time.

• We need to understand how pages link to each other.

A Piece of the Web

1. www.cs.rutgers.edu/~mlittman/courses/cs442-06/: 2, 3, 1, 2, 4, 5
2. www.cs.rutgers.edu/~mlittman/: 1, 6, 7, 10
3. paul.rutgers.edu/~babes/: 1
4. www.cs.rutgers.edu/~mlittman/courses/cs442-06/python/
5. www.cs.rutgers.edu/~mlittman/courses/cs442-06/googletest1.html: 7, 1
6. www.cs.rutgers.edu/rl3/: 8, 10
7. www.cs.rutgers.edu/~mlittman/topics/googlewhacks
9. www.cs.rutgers.edu/~mlittman/courses/cs442-06/googletest2.html: 1
Graphs

- In CS and discrete math, this kind of structure is known as a *graph*.
  - Nodes: Web pages, in this case.
  - Links: Pointer from one web page to another, in this case.
Some Graph Terms

- **source**: a node with no incoming links.
- **sink**: a node with no outgoing links.
- **path**: a list of nodes such that each adjacent pair of nodes has a link from the first to the second.
- **cycle**: a path in which the first and last node are the same.
- **connected component**: a set of nodes such that there is a path from any node to any other node in the set.
- **tour**: a cycle including all nodes in the graph.

Graphs Are Everywhere

- What are the nodes, links, paths, source, sinks, connected components of each?
- Two more definitions: A graph is **undirected** if each connected pair of nodes is connected in both directions.
- An undirected graph is a **tree** if it has no cycles.
- Is each example directed or undirected? Tree or not?
Circuit Diagram

DC motor direction controller
+9V

Control Inputs

A and B: 6N138 / 3N2022
Q1 and Q2: 2N3904
Q3 and Q5: 2N7000
Q4 and Q6: 2N100

Diodes are 1N4001

CAUTION: The max motor current rating not to exceed max. 5LPSK100 rating (1A with heat sink)

A Maze
We can represent a graph in the computer by a list of nodes, and a function that, given a node \( i \), returns the list of nodes to which \( i \) is linked.

```python
g = [[6], [1, 2, 3, 4, 5], [0, 1, 6, 7], [1], [1, 7], [0, 8], [], [2], [1]]
def links(i):
    global g
    print str(i) + " links to:"
    for k in g[i]:
        print " " + str(k)
>>> links(5)
5 links to:
 1
 7
```

A node \( j \) is reachable from a node \( i \) if there is a path that begins at \( i \) and ends at \( j \).

Let’s list all the nodes reachable from \( i \).

Any node that is reachable from a node that \( i \) is linked to is also reachable.

```python
def reachable(i):
    global g
    print str(i) + " is reachable"
    for j in g[i]:
        reachable(j)
>>> reachable(4)
4 is reachable
>>> reachable(6)
6 is reachable
0 is reachable
6 is reachable
0 is reachable
6 is reachable
0 is reachable
```
Don’t Revisit!

- What goes wrong? Once we realize we can reach some node, we should mark it as “reached” and never pursue it again.

Reachable Version 2

```python
reached = []
def reachable(i):
    global g, reached
    reached = range(len(g))
    for j in range(len(g)):
        reached[j] = False
    reachable_recursive(i)

def reachable_recursive(i):
    global g, reached
    if reached[i] == False:
        print(str(i) + " is reachable")
        reached[i] = True
        for j in g[i]:
            reachable_recursive(j)
```

```bash
>>> reachable(6)
6 is reachable
0 is reachable
8 is reachable
2 is reachable
1 is reachable
3 is reachable
4 is reachable
5 is reachable
7 is reachable
>>> reachable(4)
4 is reachable
```
• to-do list idea
• relate to word constructors...

Back to Google

• So, how does Google do it?
  I. Web crawl: download known pages, collect links to other pages, repeat
  II. Indexing: Build a giant index that associates each word with a list of pages on which it appears.
  III. Distributed search: Use lots and lots and lots of computers to do fast lookups.
Sorting Algorithms

• Another name for the lecture is “Google II”.
• Sorting is a great topic in CS:
  - relatively simple
  - extremely important
  - illustrates lots of different algorithms and
    analysis techniques

There’s more than one way to skin a cat.
What Can We Do?

- All the information is there, and we can sift through it.
- But, it’s slow and error-prone to skim through every page every time we want to find something.
- If there are $N$ words (total) on the web pages, how long would it take to sift through them each time? (Use “big O” notation.)
- How can we organize the data to simplify?

Sort, Remove Duplicates
Sorting Helps

- Phonebook, look for a last name vs. look for a first name.
- “Is there a pair that sums to 86?” Don’t have to consider all pairs.
- Is there a repeated number in the list?
- Not to mention min, max, median.

Selection Sort

- Idea is quite simple. We go through the list one item at a time.
- We keep track of the smallest item we’ve found.
- When we’re through the list, we pull the smallest item out and add it to a list of sorted items.
- We repeat until all the items have been removed.
Selection Code

def Selection(l):
    sorted = []
    while len(l) > 0:
        (smallest, rest) = findSmallest(l)
        sorted = sorted + [smallest]
        l = rest
    return sorted

def findSmallest(l):
    smallest = l[0]
    rest = []
    for i in range(1, len(l)):
        if l[i] < smallest:
            rest = rest + [smallest]
            smallest = l[i]
        else:
            rest = rest + [l[i]]
    return (smallest, rest)

Selection Sort Analysis

• How many comparisons does Selection Sort do in the worst case? Assume the list is length $N$. Hint: What song is it like? You can use “big O” notation.

• Does it matter whether the list is sorted or not?
Other Sorting Approaches

- How else can you imagine sorting?
- Fewer comparisons than $O(N^2)$?
  - bubblesort
  - counting sort
  - insertion sort
  - Shell sort

Guess Who?

- Each player picks a character.
- Players take turns asking each other yes/no questions.
- First player to uniquely identify the other player’s character wins!
Mindreader: Set Cards

A B
C D
E F G H
I J K L
M N O P

Mindreader: Set Cards

A B
C D
E F G H
I J K L
M N O P
Cross-Hatched?

Squiggle?
Insight

• Each question splits the remaining set of possibilities into two subsets (yes and no).

• We want to pick a question so that the larger of the two subsets is as small as possible.

• Half!

• How many questions?
  • \( n = 1 \), questions = 0
  • \( n = 2 \), questions = 1
  • \( n = 4 \), questions = 2
  • \( n = 8 \), questions = 3
  • \( n = 16 \), questions = 4
  • \( n \), questions = \( \lg n \).

Binary Search

• Let’s say we have a sorted list of \( n \) items.

• How many comparisons do we need to make to find where a new item belongs in the list?

• Can start at the bottom and compare until the new item is bigger.

• Maximum number of comparisons?

• One for each position: \( n \).

• We can ask better questions: bigger than the halfway mark?

• That gets us: \( \lg (n+1)! \)
### Binary Search Sort

- Using $O(\lg N)$ comparisons, can find where to insert the next item.
- Since we insert $N$ items, comparisons is $O(N \lg N)$ in total.
- Can’t quite implement it that way, though: Once we find the spot, $O(N)$ to stick it in.
- However, other algorithms are really $O(N \lg N)$.
- Hillis mentions Quick Sort and Merge Sort.

### Quicksort

- **quicksort**: Another sorting algorithm.
- **Idea**: Break the list of $n+1$ elements into the median and two lists of $n/2$. The two lists are those smaller than the median and those larger than the median.
- Sort the two lists separately.
- Glue them together: All $n$ are sorted.
**Quicksort Example**

- Original list:
  - [56, 80, 66, 64, 37, 36, 91, 48, 17, 20, 86, 89, 41, 1, 96, 12, 74]
- Median is 56; smaller: [37, 36, 48, 17, 20, 41, 1, 12]
  - bigger: [80, 66, 64, 91, 86, 89, 96, 74]
- Sort each; smaller: [1, 12, 17, 20, 36, 37, 41, 48]
  - bigger: [64, 66, 74, 80, 86, 89, 91, 96]
- Glue:
  - [1, 12, 17, 20, 36, 37, 41, 48, 56, 64, 66, 74, 80, 86, 89, 91, 96]

**But...**

- If we could find the median, the whole sorting process would be pretty easy.
- Sufficient to split anywhere in the middle half at least half the time: Still $O(n \log n)$.
- Pick a random list element. 25% of the time, it will be in the 1st quarter of the sorted list, 25% of the time in the last quarter, and 50% in the middle half.
Quicksort’s Flow

- Pick an item, any item (the “pivot”).
- Partition the list as to less (left) or greater than (right) pivot.
- Sort the two halves (recursively).

Code

def Quicksort(l):
    if len(l) <= 1: return l
    pivot = l[randint(0,len(l)-1)]
    (left,equal,right) = partition(l,pivot)
    return Quicksort(left) + equal + Quicksort(right)

def partition(l,pivot):
    left = []
    right = []
    equal = []
    for item in l:
        if item < pivot:  left = left + [item]
        if item > pivot:  right = right + [item]
        if item == pivot: equal = equal + [item]
    return (left,equal,right)
Web Search, Again

- We’ve seen two of the major steps needed to implement a web search engine:
  - gather up pages using graph search
  - index the words using sorting

- In a later lecture, we’ll talk about the last step: using more than one computer to respond quickly to millions of queries a day.

Heuristics

- Hillis makes a distinction between:
  - algorithms: “fail safe” procedures. They are guaranteed to do the job.
  - heuristics: rules of thumb. They might get pretty close to the right answer much of the time.

- In fact, we often refer to both categories as “algorithms”, but that’s ok.
Tough Problem

- Let’s return to the (known hard) problem of subset set.

- Is there a subset that sums to 433?

48, 41, 14, 46, 31, 2, 27, 12, 22, 71, 44, 63, 33, 64, 83, 28, 96, 87, 52

- One algorithm is to list all $2^n$ subsets and check each one.

- If we’re happy just getting close, we could use hillclimbing.

Hill Climbing Example

48, 41, 14, 46, 31, 2, 27, 12, 22, 71, 44, 63, 33, 64, 83, 28, 96, 87, 52

- Target = 433

- Start with a random subset: [41, 46, 2, 27, 12, 71, 44, 33, 64, 28], sum = 368.

- What number can we add to the set to get closer to the target? Including 63 gets us a sum of 431.

- Can’t get any closer to 433 by adding or removing a single number. Stuck at the top of a hill. Can start again with a different subset.
Hill Climbing Limitations

- Can get trapped in **local minimum**: need to go further to get closer.
- No guarantee that a maximum will be found.
- Can even be slow to find a local minimum!
- With the wrong scoring function, finding the right answer can be like a needle in a haystack.