Homework 4 & 5

Jeremy Wright
CSE598 - Analysis of Algorithms

October 24, 2013

Problem 1. Show how to multiple the complex numbers $a + bj$ and$c + dj$ using only three real multiplications...

Given:

$$\begin{aligned} R + jI &= (a + jb) \cdot (c + jd) \\ &= (ac - bd) + j(bc + ad) \end{aligned}$$

A common pattern in Digital Signal Processing is to break out the operations into 3 constants.

$$\begin{aligned} k_1 &= a(c + d) & &= ac + ad \\ k_2 &= d(a + b) & &= ad + bd \\ k_3 &= c(b - a) & &= bc - ac \end{aligned}$$

From these we can define the Real part as

$$\begin{aligned} \Re &= k_1 - k_2 \\ &= ac + ad - (ad + bd) \\ &= ac + ad - ad - bd \\ &= ac - bd \end{aligned}$$

The Imaginary Part as

$$\begin{aligned} \Im &= k_1 + k_3 \\ &= ac + ad + bc - ac \\ &= ab + bc \end{aligned}$$

In Python we can clearly see this uses only 3 multiplications.

```
        def complex_mul(a, b, c, d):
            k1 = a(c + d)
            k2 = d(a + b)
            k3 = c(b - a)
            return (k1 - k2) + (k1 - k3)*j
```

Problem 2. Consider in the closest pair of points problem, splitting the points along $\frac{n}{4}$ and $\frac{3n}{4}$. Is the complexity still $O(nlgn)$? Yes

Proof:

Consider the base case $n >= 4$:

$$T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + n$$

Substitute $nlgn$

$$\begin{aligned}
T(n) =& T(\frac{n}{4}) + T(\frac{3n}{4}) + n \\
T(n) \leq& c\frac{n}{4}\log_2\frac{n}{4} + c\frac{3n}{4}\log_2\frac{3n}{4} + cn \\
\leq& c\left(\frac{n}{4}\log_2 n - \frac{n}{4}\log_2 4\right) + c\left(\frac{3n}{4}\log_2 3n - \frac{3n}{4}\log_2 4\right) + cn \\
\leq& c\left(\frac{n}{4}\log_2 n - \frac{2n}{4}\right) + c\left(\frac{3n}{4}\log_2 3n - \frac{3n\cdot 2}{4}\right) + cn \\
\leq& c\left(\frac{n}{4}\log_2 n - \frac{1}{2}n\right) + c\left(\frac{3n}{4}\log_2 3 + \frac{3n}{4}\log_2 n - \frac{3n}{2}\right) + cn \\
\leq& c\frac{n}{4}\log_2 n - \frac{1}{2}cn + \frac{3nc}{4}\log_2 n - cn + cn \\
\leq& c\frac{n}{4}\log_2 n + \frac{3cn}{4}\log_2 n \\
\leq& cn\log_2 n
\end{aligned}$$

Intuitively this makes sense because we are still dividing the space into some log relatable ratio for each subproblem.

Consider dividing the space into $\sqrt{n}$ and $n - \sqrt{n}$. This problem will not remain at $n\log_2 n$ since the division of the space is not an even recurrence. Instead there is a linear adjustment in the form of $n - \ldots$

$$\begin{aligned}
T(n) &= T(n^{\frac{1}{2}}) + T(n - n^{\frac{1}{2}}) + nc \\
&\leq n^{\frac{1}{2}}\log_2 n^{\frac{1}{2}} + \left(n - n^{\frac{1}{2}}\right)\log_2\left(n - n^{\frac{1}{2}}\right)c + cn\ldots
\end{aligned}$$

Eventually we get to where we cannot resolve the logs with the roots. So let's try a greater

complexity: n^2

$$\begin{aligned}
&\leq \left(n^{\frac{1}{2}}\right)^2 + \left(n - n^{\frac{1}{2}}\right)^2 + nc \\
&\leq n + n^2 - n + nc \\
&\leq n^2 + n(1 - 1 + 1)c \\
&\leq n^2 + nc \\
&\leq \lim_{n\to\infty} \left(n^2 + nc\right) \\
&\leq cn^2
\end{aligned}$$

Thus by adding an irregular division to the sub problems we increase the complexity to recombine those subproblems.

Problem 5.1. Find the median of two databases. Use at most $O(log_2 n)$ queries.

To achieve this complexity target we must do at most a recurrence of

$$T(n) = T(\frac{n}{2}) + c \tag{1}$$

With a Base case of $T(1) = 1$, since a median of a single element is itself $\therefore$

$$T(n) = T(\frac{n}{2}) + O(1) \tag{2}$$

Intuitively, this represents an algorithm which divides the input into 2 subproblems, considers one side and spends a constant amount of time at each step. Such an algorithm may be implemented in Python:

```
1  import numpy
2
3  def median( ns ):
4      l = len(ns)
5      if l % 2 == 0:
6          return (ns[l/2] + ns[(l/2)-1])/2.0
7      else:
8          return ns[l/2]
9
10
11 def find_median(a, b):
12     l = len(a)
13     if l == 1:
14         return (median(a) + median(b)) / 2
15     if l == 2:
16         return ( max(a[0], b[0]) + min(a[1], b[1]) ) / 2
17
18     ma = median(a)
19     mb = median(b)
20
```

```
21      if ma == mb:
22          return ma
23
24      if ma < mb:
25          if l % 2 == 0:
26              return find_median(a[l/2:], b[:l/2])
27          else:
28              return find_median(a[l/2:], b[:(l/2)+1])
29      else:
30          if l % 2 == 0:
31              return find_median(a[l/2-1:], b[l/2-1:])
32          else:
33              return find_median(a[:(l/2)+1], b[(l/2):])
34
35  import random
36  for i in range(10):
37      a = [random.randint(0,10) for r in xrange(10)]
38      b = [random.randint(0,10) for r in xrange(10)]
39      c = a + b
40
41      a.sort()
42      b.sort()
43      c.sort()
44      print a, b, c
45      if median(c) == find_median(a,b):
46          print "PASS"
47      else:
48          print "FAIL"
```

This algorithm leverages subranges (called splices in Python). Lines 12 - 16 describe the various end conditions and base cases. Once complete, one must start the merging of each subrange. If the median of list a is less than the median of list b, then the median of the entire set must lie in the right half of list a, or the left half of list b.

Conversely, if list b's median is greater, then the median must lie in the left half of a.

Problem 5.5 (graduate). Design an algorithm which finds the visible lines in $O(nlgn)$ time.

The brute force of this problem, would be to compare the intersections of all lines, sorting the y values, and dropping the invisible lines. This however results in $\Omega(n^2)$.

The algorithm must then divide the input in such a way that it doesn't have to enumerate *all* intersections.

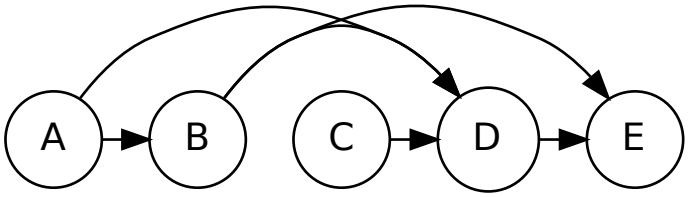
Consider:

1. Sort the list of lines by their slope.
2. Add line to the visible list.

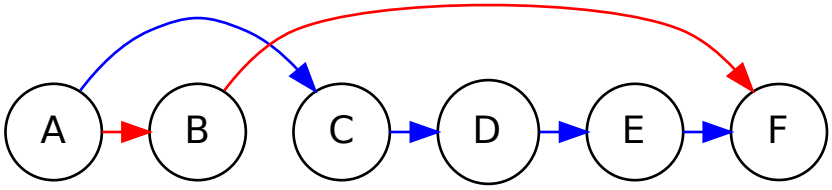
3. Starting at the beginning of the list, find all intersections with this line to the lines to its right in the list.
4. Take the line who's intersection is greatest. Add this line to the visible list.
5. Now repeat with this line, find all intersections with lines to it's right.

This algorithm sorts in O(n lg n) using something like Merge Sort. Then at each step it only considers lines to it's right in the sorted slope list. Each line with a greater y value gets added to the visible list, and becomes the next line to consider, never considering lines behind us, or already in the visible list. In this way we beat n^2 time.

Problem 6.3. Given an ordered graph G, find the length of the longest path that begins at A and ends at E.



The given algorithm is a greedy algorithm and always chooses the nearest node hoping to get to the end as late as possible. However in the example below, the blue path is the longest path, with length 4, wile the red path is length 2. The greedy algorthm does not see beyond node 2, and incorrectly follows the red path.



Dynamic programming can help here by "carefully" evaulating all possible paths. First a few constraints. The graph must be acyclic,, and have a topolocial sort, which by definition it does.

Next we use a 5 step process.

1. Define sub problems.
2. Guess a metric to minimize/maximize.
3. Relate the sub problems (recurrence)
4. Solve the recurrence, and memoize the algorithm.
5. Recombine the subproblems (verify the original problem gets solved.)

1. Define sub problems. Consider the incomming edges to the final node. The longest path must be one of those edges, so use dynamic programming to carefully try them all.

2. Guess the last edge comming into the final node v. Next recurrsively find the longest path to u and add the longer path to v.

3. Relate sub problems. Define a function $\delta(s, v)$ which is the length of the edge from the path from s to v.

$$\delta(s, v) = max\{\delta(s, u) + (u \to v).\text{weight } \forall \text{ Edges} = \text{u}\} \tag{3}$$

Time per subproblem is $\delta(s, v) = indegree(v)$. Total time

$$\sum_{v \in G} indegree(v) = \theta(E + V) \tag{4}$$

Problem 6.5. Consider the string ilovemywife.

Consider all the suffixes of this string.

define a recurrence $Q(s, f)$ where s is the starting index, and f is the closed finish index. $Q(s, f)$ returns the quality for that string. We want to maximze the quality of the sentence. So lets "carefully" try all suffixes.

1. Subproblems. suffixes of words[i:] (in python notation).
number of subproblems is n in the number of letters

2. Guess where to start the second word.
number of choices (n - 1) since the first letter must be at the first word.

3. Recurrence

$$word(i) = max\{word([j :]) + Q(i, j) \text{ for j in range(i+1, n + 1)}\} \tag{5}$$

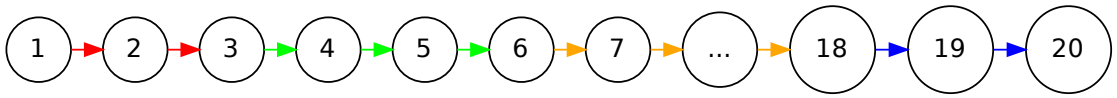
At each letter, the quality of $[i : j]$ is added to the recursive call. The suffix is then split at j, and recursively called. This measures the quality of all n suffixes of the word. Since we have overlapping subproblems, if we memoize, we can ignore the cost of the recursive calls and this results in $\theta(n)$ since there are n letters in the set of suffixes.

4. Total running time is time / subproblem * number of problems = $\theta(n^2)$.

5. Original subproblem is the maximum quality starting from the first character i.e. $word(0)$.

Problem 6.20. From the problem we can see that the function f_i is a nondecreasing function, thus it has a topological sort $\therefore$ we can apply dynamic programming since our problem is acyclic.

The goal is then to find a collection of contagious hours to spend on each class as show in the DAG below.



It is important to note that the hours are indivisible integers, since real numbers would make the dynamic program infinite. Minimally we have to divide the hours into some discrete steps to draw the DAG. Without a DAG, dynamic programming will not result in a correct algorithm.

This problem is now similar to problem 6.5 such that we are finding the divisions of contigious sequences.

1. Subproblems. Suffixes of the list of hours. Each list of hours will be fed to the function f_i and we want to maximize our grade at each point.

2. We'll start on the first hour, and try every set of hours after that. This results in the number of subproblems being $\theta(n)$

3. Recurrence

$$hours(i) = max\{hours([j:]) + f((i-j)) \text{ for j in range(i+1, n + 1)}\} \tag{6}$$

At each hour, we choose the hours until the end of the list, and calculate the expected grade for that set of hours. We then recurse on the hours left in the set. Since we have overlapping sub problems we don't have the count all the recursive calls, $\therefore$ we expect the time per subproblem to be $\theta(n)$.

4. Total running time is $\frac{time}{subproblem} \cdot$ number of subproblems $= \theta(n) \cdot \theta(n) = \theta(n^2)$

5. We don't spend any extra time combining the subproblems, thus we meet this with $\theta(n^2)$