University of Waterloo

CS 466/666

Instructor: Ian Munro DC2343 imunro@..
(Office hours: after class)

TAs: Vinayak Pathak; DC3132, vpathak@..
Yizhe Zeng; DC2555A, y29zeng@..
(Office hours to be announced)

For most information, see course page

https://www.student.cs.uwaterloo.ca/~cs466/

Will use Piazza as newsgroup

Requirements

Details on course page:

About 10 short assignments: 25%

Midterm: 25%

Final:50%

For grad students (CS666) also a project/ paper review: 25% (the rest *.75)

Course Outline

- Review of key points in CS 341
- Randomized Algorithms
- Amortized Analysis
- Lower Bounds
- Approximation Algorithms (NP-complete problems)
- Parameterized Complexity (Special cases of NPC)
- Online Algorithms

Approaches from CS 341

- Divide and Conquer
- Dynamic Programming
- Greed

We'll look at an example or more of each

Divide & Conquer: Master Thm +

Thm: Let $a \geq 1$, $b > 1$, $c \geq 0$ be constants, $f(n) = n^c$ and T(n) be defined on nonnegative integers by the recurrence:

$$T(n) = a(T(n/b) + f(n)$$

{ we take n/b as $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$ }

Then:

if (1) $b^c > a$: $T(n) = \Theta(n^c)$ last "fixup" domnates

(2) $b^c = a$: $T(n) = \Theta(n^c \lg n)$ balance

(3) $b^c < a$: $T(n) = \Theta(n^{\log_b a})$ small problems dominate

Examples

Binary search

Mergesort

Large number multiplication (Karatsuba)

Strassen matrix multiplication

Median algorithm (BFPRT)

Karatsuba

Large n precision, number multiplication in $O(n^{\lg 3})$ time ($\lg 3 \approx 1.59$) as:

Let $x = (p2^{n/2} + q)$ and $y = (r2^{n/2} + s)$

then

$xy = (p2^{n/2} + q)(r2^{n/2} + s)$

$= pr2^{n} + (ps + qr)2^{n/2} + qs$

$= pr(2^{n}+2^{n/2})-(p-q)(r-s)2^{n/2}+qs(2^{n/2}+1)$ (check it)

In fact we can compute these in three n/2 precision multiplications, so

$T(n) = 3T(n/2) + n$

and $T(n) = O(n^{\lg 3})$

Usual Matrix Multiplication

Standard method: C= AB

$c_{ij} = \Sigma\, a_{ik}\, b_{kj}$

First consider multiplying 2 by 2 matrices

$$\begin{pmatrix} \mathbf{c_{11}} & \mathbf{c_{12}} \\ \mathbf{c_{21}} & \mathbf{c_{22}} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{a_{11}} & \mathbf{a_{12}} \\ \mathbf{a_{21}} & \mathbf{a_{22}} \end{pmatrix} \begin{pmatrix} \mathbf{b_{11}} & \mathbf{b_{12}} \\ \mathbf{b_{21}} & \mathbf{b_{22}} \end{pmatrix}$$

Usual method uses 8 multiplications (and 4 additions).

Even recursing, we would get

$T(n) = 8\, T(n/2) + O(n^2)$

Hence a $O(n^3)$ method

Strassen Matrix Multiplication

$m_1 = (a_{12} - a_{22})\ (b_{12} - b_{22})$

$m_2 = (a_{11} + a_{22})\ (b_{11} + b_{12})$

$m_3 = (a_{11} - a_{21})(b_{11} + b_{12})$

$m_4 = (a_{11} + a_{12})\ b_{22}$

$m_5 = a_{11}(b_{12} - b_{22})$

$m_6 = a_{22}(b_{21} - b_{11})$

$m_7 = (a_{21} + a_{22})b_{11}$ then

$c_{11} = m_1 + m_1 - m_1 + m_1$

$c_{12} = m_4 + m_5$

$c_{21} = m_6 + m_7$

$c_{22} = m_2 - m_3 - m_5 + m_7$

Uses 7 (not 8) multiplications (and 17 additions)

So $T(n) = 7\ T(n/2) + \Theta(n^2)$

Hence a $\Theta(n^{\lg 7})$ method ($\lg 7 = 2.81..$)

Linear Median Algorithm (BFPRT)

If n is "small"; sort and select the k^{th}

Else {

sort elements in n/c "columns" of length c (c odd)

recursively find median of column medians

determine where median of median fits in each column
(so have rank of median)

(Those falling on "wrong" side of median cannot be k^{th})

do appropriate recursive selection on the others

}

Calculations …

Optimal Binary Search Trees: Dynamic Programming

Searching under the comparison model (<,=,> branching)

Binary search: lg n upper and lower bounds also in "expected case" (probability of search same for each element)

With some balanced binary scheme, updates also in O(lg n)

But what if some elements are requested more than others?

Start with stochastic model: Given a set of n keys, K =<k_1.. k_n> with independent probabilities of access, p_i.

How can we organize a search structure to minimize the expected number of comparisons for a search?

Clearly this is a binary search tree, though in general far from balanced.

How do we find the optimal tree?

Optimal Binary Search Trees

Try a few top down heuristics, and easily get non-optimal trees.

We need some definitions:

k_i : i^{th} largest of key value i=1..n

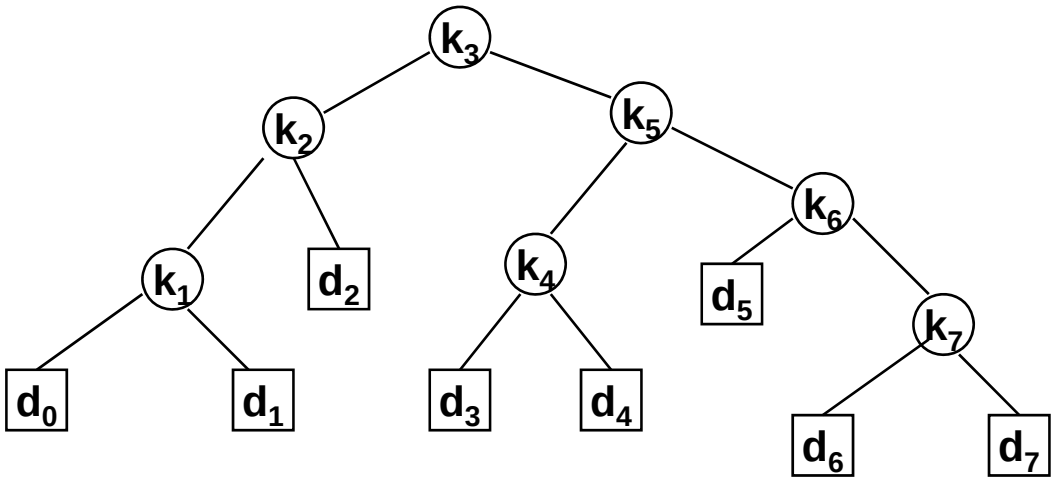
d_i : dummy leaf after k_i, before k_{i+1} i=0..n

p_i : probability of a request for k_i

q_i: probability of a request for an element after p_i and before p_{i+1}.

w(i,j): probability of element in OPEN INTERVAL (k_{i-1},k_{j+1}), so

$w(i,j)=\sum_{k=i,.j}p_k + \sum_{k=i-1..j}q_k$; w(1,n)=1 {find all w(i,j) in $O(n^2)$ time}



The Dynamic Program

Expected cost of a search:

E[search in T] = $\sum(\text{depth}_T(k_i) +1)p_i + \sum(\text{depth}_T(d_i) +1)q_i$

So ..

Compute

e[i,j] = cost of optimal i,j tree

= q_{i-1} if j=i-1 and

= $\min_{i \le r \le j}\{e[i,r-1]+e[r+1,j]+w(i,j)\}$ if $i \le j$

Also keep track of the root as r[i,j]

r[i+1,i] = d_i ; r[i,i] = k_i ;

Otherwise r[i,j] determined by e[i,j] calculation

This gives a straightforward dynamic program, which we can do with a loop r=i,..j and recursive calls, of three loops for $\Theta(n^3)$ time, and $\Theta(n^2)$ space.

Improvement

Note: loop to compute {r,e}[i,j] goes all the way… i to j.
Is all this necessary?

Lemma: r[i,j] cannot precede r[i,j-1] or follow r[i+1,j]. {Omit proof}

So modify the inner loop
Change "r = i..j" to "r = r[i,j-1]..r[i+1,j]"
Look at runtime; series telescopes

Theorem: The optimal binary search tree can be determined in $\Theta(n^2)$ time and $\Theta(n^2)$ space.

This is the best known algorithm, indeed there is no known polynomial time, $o(n^2)$ space method.

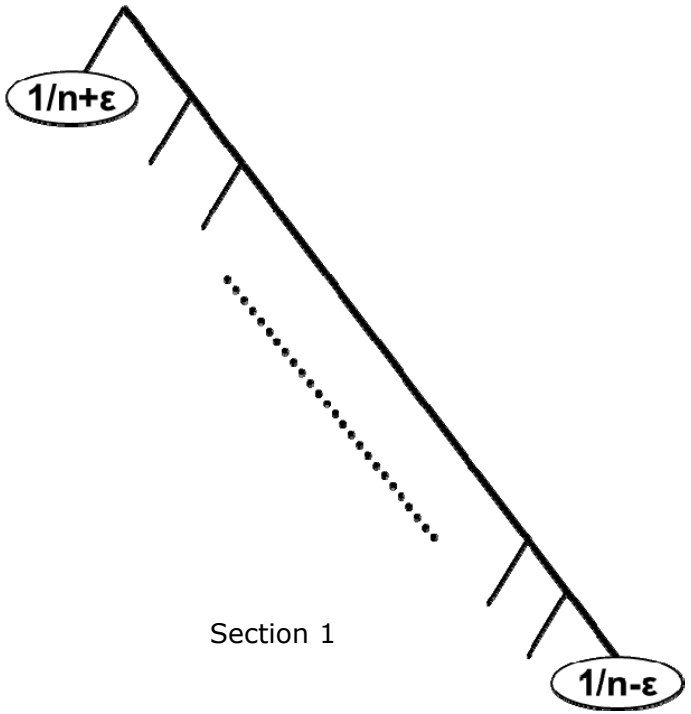
Good Trees in Less Space

Suppose we don't have $\Theta(n^2)$ space.

How can we get a good tree?

Greed !!!

First Attempt: Choose root as key with greatest probability

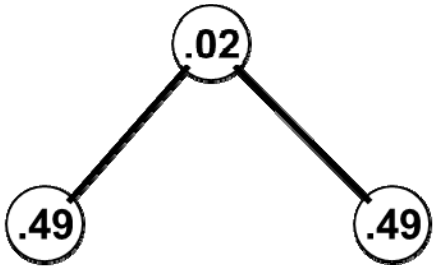


Better Greed

Choose root to balance the weights on either side as well as possible. There are a few (picky) options:

MinMax: Minimizes the weight of largest subtree

"Balance": Choose root to make subtree wts as close as possible



Not optimal; but perhaps, not bad.

Naïve algorithm is $\Theta(n^2)$ (worst case), but O(n) space.

How can we make it faster?

A Faster Greed Algorithm

Given keys in order, and probs: p_i, d_i

Let L_i = probability of being left of k_i

For root of tree in range [st,fin], find, by binary search, key with L_i below and L_{i+1} above $(L_{st}+L_{fin+1})/2$

Hence an O(n lg n) method.

Can we improve this?

Yes … note the method is linear if you always "get lucky" with "split" near the middle.

So … Start at with one comparison in the middle. Then move to the "side still in" and double your way toward the middle, till desired element "bracketed", finish with binary search.

Runtime

Cost of discovering split point:

$\leq$ 1+ 2 lg v, v is #keys from near end.

Thm: The algorithm runs in O(n) time. i.e. the splits have an amortized cost of O(1).

Proof sketch:

Try induction: $T(n) \leq \alpha n$? So try $T(n) \leq \alpha n - \beta \lg n$

Tune constants for the base cases

Basic recurrence:

$T(n) \leq T(a) + T(n-a-1) + 2 \lg a \quad \{1<a\leq n/2\}$

Substitute: $T(n) = \alpha a - \beta \lg(a) + \alpha n - \alpha a - \alpha - \beta \lg(n - a - 1) + 2 \lg a$

We require

$\beta \lg(a) + \beta \lg(n-a-1) + \alpha > \lg n$ {when a<n/2}

This is fine when a is large, α and β have to be tuned to handle the small values of a.

Efficacy of Solutions

The approximation method is rather good. Define:

- $P = \sum p_i$
- H, the entropy of a distribution, as $H(p_1..p_n,q_0..q_n) = -\sum p_i \lg p_i + \sum q_i \lg q_i$ {Note H is maximized when all probabilities are the same}
- C_{opt} and C_{approx} as tree costs

Then

Thm: $H - P \lg(eH/2P) \leq C_{opt} \leq C_{approx} \leq H+2 - P$

i.e. optimal and approximate tree have costs with lg H of optimal.

Proof: Omitted

But what it probabilities change… or we don't know them?

Could count accesses and update optimal tree based on changing probabilities.

{this has been done for Huffman codes}

Or

Recall linear search and the "move to front" heuristic. Assume list starts empty and element put at the end the first time it is requested

Thm (from CS 240): The cost of a sequence of searches under the move to front heuristic is within a factor of 2 of that of the optimal (static) order.

Indeed

Thm: The amortized cost of a search under move to front is at most twice the optimal we could get if we knew the sequence and updated the list. {Off line updates work by swapping adjacent items}