CS270 Combinatorial Algorithms & Data Structures **Spring 2004**

Lecture 9: 2.17.04

Lecturer: Satish Rao *Scribe: Adam Chlipala*

9.1 Linear programming wrap-up

9.1.1 Seidel's algorithm

Given the linear program $Ax \leq b; \max c$ with d variables and n constraints:

Algorithm 1 $Seidel(A, b, c)$

1: **if** there is only one variable **then**
2: Treat all inequalities in $Ax \leq b$ as equalities and solve for x in each.
3: Return the point out of these that maximizes c.
4: **end if**
5: **if** there are as many constraints as variables **then**
6: Solve the linear system $Ax = b$ using Gaussian elimination and return the resulting point.
7: **end if**
8: Pick a constraint $h : a_i x \leq b_i$ uniformly at random.
9: $x^* \leftarrow Seidel(A - a_i, b - b_i, c)$
10: **if** x^* satisfies h **then**
11: Return x^*
12: **else**
13: $A', b' \leftarrow$ substitution of the equality $a_i x = b_i$ in $Ax \leq b$
14: Return $Seidel(A', b', c)$
15: **end if**

9.1.1.1 Correctness

Lines 1 through 7 solve the particularly simple cases that have only one variable or as many constraints as variables. The remaining code recurses until one of these special cases is reached.

Clearly line 11 will only return correct values: We inductively assume that x^* is the best solution without the chosen h. Reincluding h could only restrict the solution space, but x^* is verified to satisfy h, so it must be the solution to the original problem. Figure 9.1 illustrates this case.

If line 14 is reached, then x^* did not satisfy h. This means h is one of the "important" tight constraints that determines the solution. Therefore, we know that the solution lies on h's hyperplane, so $a_i x = b_i$ for the true solution x. This equation may be simplified to one with only one of the variables on the lefthand side, allowing the substitution of the righthand side for that variable in A and b, lowering the dimensionality of the problem. We know that the new system is equivalent to the old, so, inductively, the return value of the recursive invocation is the correct solution. Figure 9.2 illustrates this case.

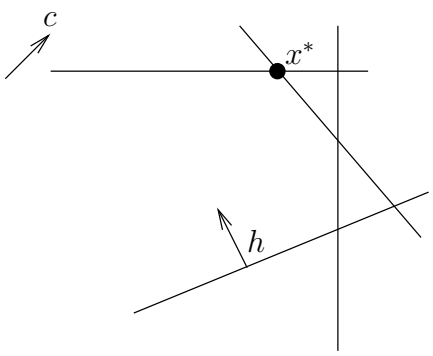


Figure 9.1: Removing a nontight constraint

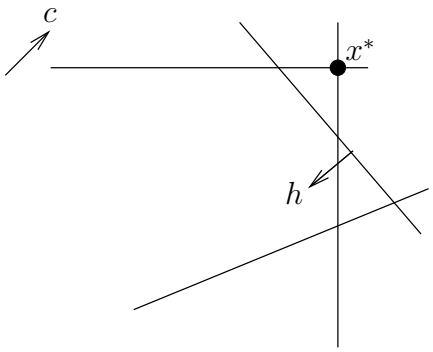


Figure 9.2: Removing a tight constraint

9.1.1.2 Running time

Denote by $T(n, d)$ the expected running time of Seidel's algorithm on an input with d variables and n constraints. Aside from the base cases, *Seidel* will always make one recursive call with a system of one fewer constraint. If the randomly chosen h is a tight constraint, it will also make a recursive call with one fewer variable. There are exactly d tight constraints, so there is a $\frac{d}{n}$ chance of this recursive call occurring.

Thus, considering that the time for recursion will dominate the time spent by the base cases, we arrive at this recurrence:

$$T(n, d) = T(n-1, d) + \frac{d}{n} T(n, d-1)$$

There are simple methods that can be used for solving common recurrences, but they don't apply to this one! It turns out that a solution for this one is:

$$T(n, d) = O(d!n)$$

Thus, if we hold d fixed, Seidel's algorithm runs in expected linear time.

9.1.2 Summary of linear programming

We've looked at these algorithms:

Simplex While it performs very well in practice, it doesn't run in polynomial time. However, it has recently been proven that a slight perturbation of any family of linear programs leads to a family for which Simplex runs in polynomial time.

Karmarkar's interior point method Both polynomial time and efficient in practice.

The ellipsoid method The first polynomial time algorithm discovered, but very inefficient in practice. However, it generalizes to nonlinear systems and is useful in reasoning about many kinds of problems.

Seidel's algorithm Randomized algorithm that is efficient for problems with low fixed dimensionalities.

Linear programming is very useful for game theoretic problems. Many, many other interesting problems can be reduced to linear programming problems.

9.2 Recurrences

9.2.1 An example recurrence

Consider this recurrence:

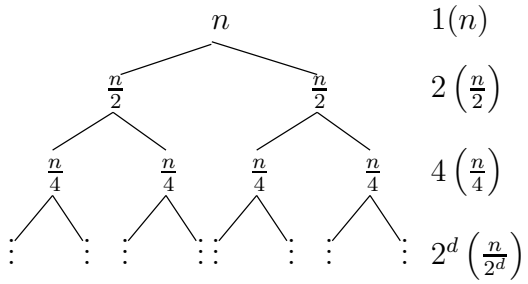
$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ T(1) &= 1 \end{aligned}$$

We've all seen this one before for algorithms like merge sort, so the solution $T(n) = O(n \log n)$ comes to mind. We can verify this by "guessing" $T(n) = an \log n$ for some a and verifying that the solution has the

right properties:

$$\begin{aligned} 2T\left(\frac{n}{2}\right)+n &= 2a\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right)+n \\ &= an(\log n-\log 2)+n \\ &= an\log n-an+n \\ &= an\log n-(a-1)n \\ &\leq an\log n \\ &= T(n) \end{aligned}$$

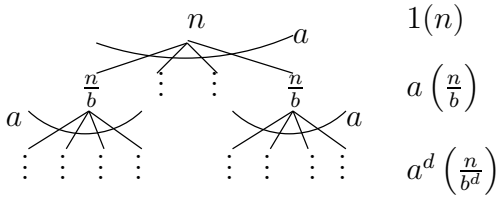
However, there's a simple procedure for solving many divide-and-conquer recurrences directly. Consider the recursion tree for an algorithm with running time described by the T above, with every node labeled with the amount of work done in it as opposed to the recursive calls it makes:



The numbers in the righthand column give the total work for each level of the tree, with the last row giving the general term for the sum of level d. It is clear that the total work on each level is n. The number of levels is $O(\log n)$. Thus, summing up the total amount of work, we arrive at a $O(n\log n)$ bound.

9.2.2 General divide-and-conquer recurrences

This sort of reasoning generalizes to recurrences of the form $T(n)=aT\left(\frac{n}{b}\right)+n$ for any a,b:



For the general form, we have a branching factor of a with $\frac{n}{b^a}$ done at every node at level d. Thus, we have:

$$T(n)=n\sum_{i=0}^{\log_b n}\left(\frac{a}{b}\right)^i$$

If $a<b$, then $\frac{a}{b}<1$, and the sum is a prefix of a convergent geometric series, making the sum $O(1)$ for fixed a and b. This means $T(n)=O(n)$.

If $a = b$, then we have the same sort of recursion as in the original example, yielding $T(n) = O(n \log n)$.

Finally, if $a > b$, then $\frac{a}{b} > 1$, and the last term of the sum dominates asymptotically, so $T(n) = O\left(n \left(\frac{a}{b}\right)^{\log_b n}\right)$. The time expression simplifes to $a^{\log_b n}$, and using the trick of switching the base and log parameter, we have $T(n) = O(n^{\log_b a})$.

9.2.3 The Master's Theorem

Now consider the recurrence $T(n) = aT\left(\frac{n}{b}\right) + n^2$.

By using the substitution $m = n^2$, we can use the result from the last section to solve this recurrence. Let $S(m) = T(n)$. We obtain:

$$S(m) = aS\left(\frac{m}{b^2}\right) + m$$

Now the last section's result can be used directly to find S. From there, we obtain $T(n) = S(n^2)$.

Generalizing this approach to arbitrary exponents for n, we can obtain the well-known Master's Theorem.

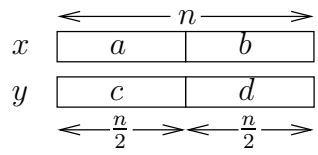
This approach also works for recurrences that make recursive calls with different size parameters. For example, for

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

the recursion tree can be seen to look a lot like that for the first recurrence we considered. The branching factor is 2, and the depth is $\log_{2/3} n$, leading to another $T(n) = O(n \log n)$ conclusion.

9.3 Clever multiplication

Standard "grade-school" multiplication of two n-bit integers takes $O(n^2)$ time, but it is possible to do better. Consider this decomposition of two numbers x and y to be multiplied:



We can treat the multiplication of x and y as multiplication of two two-digit base-$2^{n/2}$ numbers. In other words:

$$xy = ac2^n + bc2^{n/2} + ad2^{n/2} + bd$$

Now these sub-multiplications may be decomposed in the same way, down to the point where we only wish to multiply single-digit numbers. This leads to the recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Unfortunately, by the Master's Theorem, this only gives $T(n) = O(n^2)$, which is no better than what the naive algorithm achieves. However, we can use a clever trick invented by five-year-old Gauss to do better.

We need only perform three multiplications: $(a+b)(c+d)$, bd, and ac. The last two give us two of the four products we need to find the final result, with ac shifted by n in constant time to obtain $ac2^n$. By subtracting bd and ac from the first product, we obtain $bc+ad$. Shifting this by $\frac{n}{2}$ in constant time, we obtain $bc2^{n/2}+ad2^{n/2}$. Now we can add all the terms we've obtained so far to obtain the answer.

With this strategy, only 3 multiplications are performed per recursion tree node, changing the recurrence to

$$T(n) = 3T\left(\frac{n}{2}\right) + n$$

So now $T(n) = O(n^{\log_2 3})$, so $T(n) = O(n^{1.6})$, which makes a significant difference in practice. It's possible to do even better than this with similar techniques. Karatsuba's algorithm for multiplying matrices is based on this idea.

9.4 Computing medians

Definition 9.1 *The **median** of a sequence S is the $\frac{|S|}{2}$th lowest element of the sequence.*

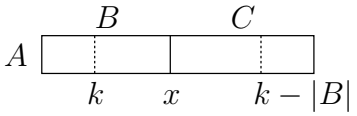
One obvious way to find a median of a sequence is to first sort it in $O(n \log n)$ time and then read the middle element of the sequence in $O(1)$ time. However, it is possible to do better. We'll use a divide-and-conquer approach that solves a more general problem:

Definition 9.2 *The **selection problem** is to, given a sequence S and index k between 0 and $|S|-1$, find the kth lowest element of S.*

Algorithm 2 *Select*(A, k)

```
if k = 0 then
  Return min(A).
else
  Choose x ∈ A uniformly at random.
  B ← ⟨y ∈ A : y ≤ x⟩
  C ← ⟨y ∈ A : y > x⟩
  if k < |B| then
    Return Select(B, k).
  else
    Return Select(C, k − |B|).
  end if
end if
```

The correctness of the algorithm is best argued with a diagram:



A sequence A is shown divided into the B and C that *Select* would form for a chosen x, with B and C considered to be sorted. If the $k < |B|$, then $A[k]$ must be in B, since B contains at least the $k+1$ lowest

elements of A. Furthermore, B must be the kth lowest element of B as well, so the first recursive call will always yield the correct return. Similarly, if $k \geq |B|$, then $A[k]$ must end up in C, and it must be the $(k - |B|)$th lowest, since the $|B|$ lowest elements of A have been moved to B. Thus, *Select* is correct.

Letting T be the running time of *Select* on an A of length n. We have:

$$E[\,T(n)] \leq E[\,T(\gamma n)] + n$$

where γ is the proportion of A's elements that end up in the subsequence that's used in the recursive call.

We might think we can proceed with

$$\begin{aligned} E[\,T(n)] &\leq E[\,T(\gamma n)] + n \\ &\leq E[\,T(E[\,\gamma\,]\,n)] + n \\ &\leq \sum_{\beta} \Pr[\gamma = \beta] E[\,T(\beta n)] + n \end{aligned}$$

but the second step is invalid. Intuitively, we see that $E[\,\gamma] \sim \frac{3}{4}$. If we could substitute this in the recurrence, we would get $E[\,T(n)] \leq E\big[\,T\left(\frac{3}{4}n\right)\big] + n$ and find that $T(n) = O(n)$ is a solution.

However, we can actually obtain this result formally through a slight modification to the algorithm.

Algorithm 3 *Select*2(A, k)

```
if k = 0 then
  Return min(A).
else
  repeat
    Choose x ∈ A uniformly at random.
    B ← ⟨y ∈ A : y ≤ x⟩
    C ← ⟨y ∈ A : y > x⟩
  until min{|B|, |C|} > |A|/4
  if k < |B| then
    Return Select2(B, k).
  else
    Return Select2(C, k − |B|).
  end if
end if
```

With an antagonistic random number generator, the new algorithm may never terminate, but most likely it will terminate, and we know that γ will never be higher than $\frac{3}{4}$. Therefore:

$$E[\,T(n)] \leq E\left[\,T\left(\frac{3}{4}n\right)\right] + E[\,d\,]\,n$$

where d is the number of iterations of the loop required before a suitable x is found. Half of the x values are suitable, so $E[\,d] = 2$, and we obtain $E[\,T(n)] = O(n)$ by the Master's Theorem.

The running time of *Select*2 is actually an upper bound on the running time of *Select*. All of the work for these algorithms is done in partitioning input sequences. When *Select* chooses an x such that $\min\{|B|, |C|\} > \frac{|A|}{4}$, then it behaves just like *Select*2 for that step. If $\min\{|B|, |C|\} \leq \frac{|A|}{4}$ and k is in the smaller subsequence, then *Select* can only do better than *Select*2 does, since *Select*2 makes its recursive call with a larger sequence. In the final case, $\min\{|B|, |C|\} \leq \frac{|A|}{4}$ and k is in the larger subsequence. Here, *Select*'s recursive call uses a larger sequence than *Select*2's, but *Select*2's first repetition of its loop when *Select*'s choice of x is picked

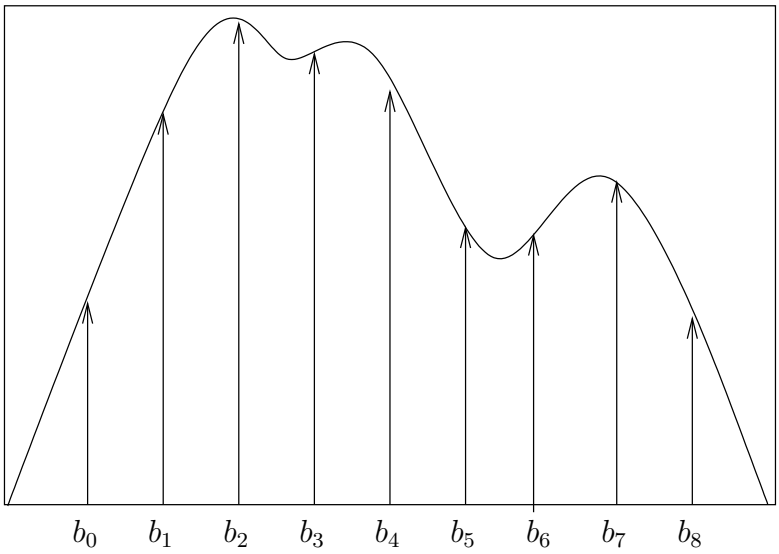


Figure 9.3: Linear system B

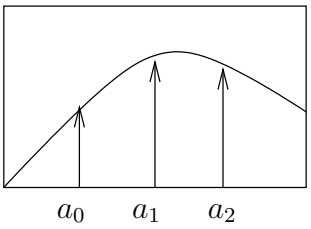


Figure 9.4: Linear system A

first does the same work as *Select*'s recursive call does. In effect, the work done by *Select* is pushed up into the loops of *Select*2 until the property that recursive calls are only made on arrays no larger than $\frac{3}{4}$ of the input sequence size is met. After such a transformation, every execution of *Select* becomes an execution of *Select*2, and so the $O(n)$ expected time bound applies.

9.5 The Fast Fourier Transform

9.5.1 Motivation

Consider a continuous linear system B like that in Figure 9.3. The x axis represents time and the y axis represents something like the energy in the system at that time. The continuous system is approximated with a discrete system through sampling, represented by arrows in the figure.

When an instant force is applied to B, the "shape" of the resulting system is B's shape, but the magnitude is proportional to the magnitude of the force. When a continuous force like A in Figure 9.4 is applied, the result is the superposition of the different results of instant forces within A. Sampling A discretely as well, we arrive at the decomposition shown in Figure 9.5 for the system resulting from applying A to B. The

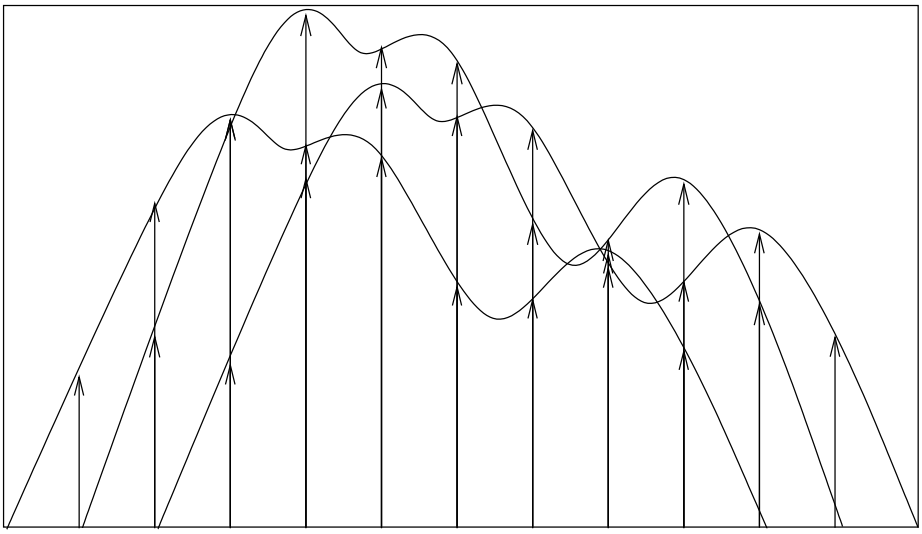


Figure 9.5: Decomposition of the result of applying A to B

energy in the system at each time is the sum of the energies of the different parts of the decomposition active at that time. Each decomposition corresponds with an instant force within A, and each starts one time unit later than the last.

Now consider general systems A and B of the types considered above, sampled over t time units. Denote by C the system resulting from the application of A to B. There is a simple pattern in the equations for the intensities of C at different time unit:

$$\begin{aligned} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\ &\vdots \quad \vdots \quad \vdots \end{aligned}$$

The naive algorithm for calculating C takes $O(t^2)$ multiplications, one for each pair of intensities between A and B. It turns out that we can do better than this.

Consider the system A as a polynomial $A(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{t-1} x^{t-1}$. Define $B(x)$ analogously, and let $C(x) = A(x) \cdot B(x)$. Upon inspecting the polynomial C, it becomes clear that its coefficients are exactly the c_i values that we're looking for.

Now that we've reduced the problem to multiplication of polynomials, we can use the Fast Fourier Transform algorithm to find C in $O(t \log t)$.

9.5.2 Outline of the algorithm

There are two main strategies used for representing polynomials. The one we've used so far represents a degree n polynomial by its $n + 1$ coefficients. Alternatively, a degree n polynomial is also defined by $n + 1$ distinct points.

This means we can represent $A(x)$ by $(x_0, A(x_0)), ..., (x_{2t-1}, A(x_{2t-1})$ for distinct $x_0, ..., x_{2t-1}$. If we represent $B(x)$ in the same manner with the same x_i's, then $C(x)$ is $(x_0, A(x_0) \cdot B(x_0)), ..., (x_{2t-1}, A(x_{2t-1}) \cdot B(x_{2t-1}))$. When we have the new forms of A and B, the new form of C can be found with $O(t)$ multiplications.

Motivated by this possibility, we'd like to convert between polynomial representations as necessary to take advantage of it. We can propose the following shell for an algorithm:

Algorithm 4 $FFT(A, B)$

1: Choose distinct $x_0, ..., x_{2t-1}$.
2: **Evaluation:** Calculate $A(x_i)$ and $B(x_i)$ for each $0 \leq i < 2t$.
3: Calculate $C(x_i)$ for each $0 \leq i < 2t$ by multiplying $A(x_i) \cdot B(x_i)$.
4: **Interpolation:** Determine C from its values at the x_i's.

Lines 1 and 3 take $O(t)$ time. To beat $O(t^2)$ overall running time, we'll need to find ways to do lines 2 and 4 in $o(t^2)$ time.

One way to think about evaluation is as computing the result of the following matrix multiplication:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{t-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{t-1} & x_{t-1}^2 & \cdots & x_{t-1}^{t-1} \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_{t-1} \end{bmatrix} = \begin{bmatrix} A(x_0) \\ \vdots \\ A(x_{t-1}) \end{bmatrix}$$

Interpolation can be viewed very similarly. Left multiplying both sides of this equation by the inverse of the matrix yields an equation for finding C from enough points through another matrix multiplication.

The naive way of performing these multiplications takes $O(t^2)$ time. However, if we can find a better way that takes advantage of the special structure of the matrix, then we can beat the naive polynomial multiplication algorithm. In fact, the FFT does just this.

To be continued....