Probability and Computation

Harchol-Balter & Sutner

Carnegie Mellon University

Spring 2012

Outline

Algorithm Performance

There are several measures that determine the quality of an algorithm: time and space complexity are the most important ones, but in the Real World there is also correctness, maintainability, portability and so forth.

Given an algorithm A and in instance I define the running time of A on I as

$$T_A(I) = \#\text{ steps in execution of } A \text{ on } I.$$

We will focus on running time and similar measures.

Worst Case Running Time

As defined, $T_A(I)$ is rather cumbersome, it is often better to focus on the worst case complexity

$$T_A(n) = \max\big(\, T_A(I) \mid I \text{ has size } n \,\big)$$

Size here is defined in a natural way: the number of bits in the input, though sometimes we may think of integers as having size 1 (logarithmic versus constant cost).

The problem with worst case complexity is that a few bad inputs may push $T_A(n)$ away from what is typically observed.

Example

quicksort, simplex algorithm.

Average Case Running Time

This suggests to average over all inputs:

$$T_A^{\text{avg}}(n) = \sum_{|I|=n} p_I \cdot T_A(I) = \mathsf{E}[T_A(I_\sigma)]$$

where p_I is the probability of receiving input I from the collection of all inputs of size n, and I_σ indicates the corresponding probability distribution.

Note that in a realistic setting σ may well be unknown: it may not be clear what instances in the sample space are more or less likely to appear in actual input.

Probabilistic Algorithms

If we give up on the notion of a deterministic algorithm we naturally run into questions relating to probability. Most importantly:

- What is the likelihood that a nondeterministic algorithm returns the right answer?
- What is the likelihood that a nondeterministic algorithm terminates quickly?

Since truly nondeterministic algorithms don't come with probability bounds we refer to the ones that do as probabilistic algorithms or randomized algorithms.

For the moment we ignore the question where the algorithms might get their random bits from.

Averaging

Note that for a probabilistic algorithm the running time may well depend on the random choices made. Hence we can talk about the average case running time for a single input I.

Given some input I, the algorithm A makes a sequence of random choices during execution. We can think of these choices as represented by a choice sequence $C \in \mathbf{2}^\star$.

Given I and C, the algorithm behaves in an entirely deterministic fashion: $A(I;C)$. So we have

$$T_A^{\text{avg}}(I) = \mathsf{E}[T_A(I;C_\sigma)]$$

where σ indicates the probability distribution of the choice sequences.

Two Important Classes

Monte Carlo Algorithms

These algorithm are always fast but may produce wrong answers with a certain small probability.

Typical examples are randomized decision algorithms.

Las Vegas Algorithms

These algorithm are always correct but may produce long running times with a certain small probability.

"Long" should be taken with a grain of salt here, it could mean the difference between polynomial and exponential time, but it could also mean the difference between $n \log n$ and n^2.

Digression: Harmonic Numbers

Definition

The nth *harmonic number* H_n is defined by

$$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$$

First few values:

$$1, \frac{3}{2}, \frac{11}{6}, \frac{25}{12}, \frac{137}{60}, \frac{49}{20}, \frac{363}{140}, \frac{761}{280}, \frac{7129}{2520}, \frac{7381}{2520}$$

From calculus, the series $\sum 1/n$ diverges, so $H_n \to \infty$ as $n \to \infty$. But divergence is glacially slow: $H_{10000} \approx 9.79$. The numerator of this fraction has 4346 digits, and the denominator has 4345.

Two Useful Equations

Surprisingly, getting a good asymptotic formula for H_n is rather hard.

$$H_n = \ln n + \gamma + \frac{1}{2n} + O(n^{-2})$$

where $\gamma \approx 0.5772156649015328$ is the *Euler constant*. The logarithm here is understood to be the natural logarithm.

A useful sum involving harmonic numbers is

$$\sum_{k=1}^{n} H_k = (n+1) \cdot H_n - n = \Theta(n \log n)$$

- Algorithms and Probability

2. Circuit Evaluation

- Yao's Minimax Principle

- Order Statistics

- More Randomized Algorithms

Minimax Trees

Here is a highly simplified model of a game tree: we only consider Boolean values $\mathbf{2} = \{0, 1\}$ and represent the two players by alternating levels of "and" and "or" gates (corresponding to min and max).

More precisely, define Boolean functions $T_k : \mathbf{2}^{4^k} \to \mathbf{2}$ by

$$T_1(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$$

$$T_{k+1}(\boldsymbol{x}) = T_k(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4)$$

where $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4)$.

Lazy Evaluation

We are interested in evaluating the circuit T_k given an assignment $\alpha : \boldsymbol{x} \to \mathbf{2}$ without reading all the values $\alpha(x_i)$.

The canonical algorithm reads variables from left to right and recursively descends from the root to the leaves. We avoid evaluation of the right subtree whenever possible. For example, in T_1, $x_1 = x_2 = 0$ already forces output 0 and we do not need to read x_3 or x_4.

Think of choosing a truth assignment for x_1, x_2, x_3, x_4 at random and define discrete random variables

$$R = \text{ output value}$$
$$S = \#\text{ variables read}$$

Augmented Truth Table

x_1	x_2	x_3	x_4	R	S
0	0	0	0	0	2
0	0	0	1	0	2
0	0	1	0	0	2
0	0	1	1	0	2
0	1	0	0	0	4
0	1	0	1	1	4
0	1	1	0	1	3
0	1	1	1	1	3
1	0	0	0	0	3
1	0	0	1	1	3
1	0	1	0	1	2
1	0	1	1	1	2
1	1	0	0	0	3
1	1	0	1	1	3
1	1	1	0	1	2
1	1	1	1	1	2

Essential Part

x_1	x_2	x_3	x_4	R	S
0	0	.	.	0	2
0	0	.	.	0	2
0	0	.	.	0	2
0	0	.	.	0	2
0	1	0	0	0	4
0	1	0	1	1	4
0	1	1	.	1	3
0	1	1	.	1	3
1	.	0	0	0	3
1	.	0	1	1	3
1	.	1	.	1	2
1	.	1	.	1	2
1	.	0	0	0	3
1	.	0	1	1	3
1	.	1	.	1	2
1	.	1	.	1	2

Some Probabilities

Here is the PMF:

$R\backslash S$	1	2	3	4
0	0	$1/4$	$1/8$	$1/16$
1	0	$1/4$	$1/4$	$1/16$

$$\mathsf{E}[R] = 9/16 \approx 0.56$$
$$\mathsf{E}[S] = 21/8 \approx 2.63$$
$$\mathsf{Pr}[\, S \mid R = 0\,] = 3/2$$
$$\mathsf{Pr}[\, S \mid R = 1\,] = 9/8$$

Table for Biased Input

x_1	x_2	x_3	x_4	R	Pr
0	0	0	0	0	q^4
0	0	0	1	0	pq^3
0	0	1	0	0	pq^3
0	0	1	1	0	p^2q^2
0	1	0	0	0	pq^3
0	1	0	1	1	p^2q^2
0	1	1	0	1	p^2q^2
0	1	1	1	1	p^3q
1	0	0	0	0	pq^3
1	0	0	1	1	p^2q^2
1	0	1	0	1	p^2q^2
1	0	1	1	1	p^3q
1	1	0	0	0	p^2q^2
1	1	0	1	1	p^3q
1	1	1	0	1	p^3q
1	1	1	1	1	p^4

Biased Input

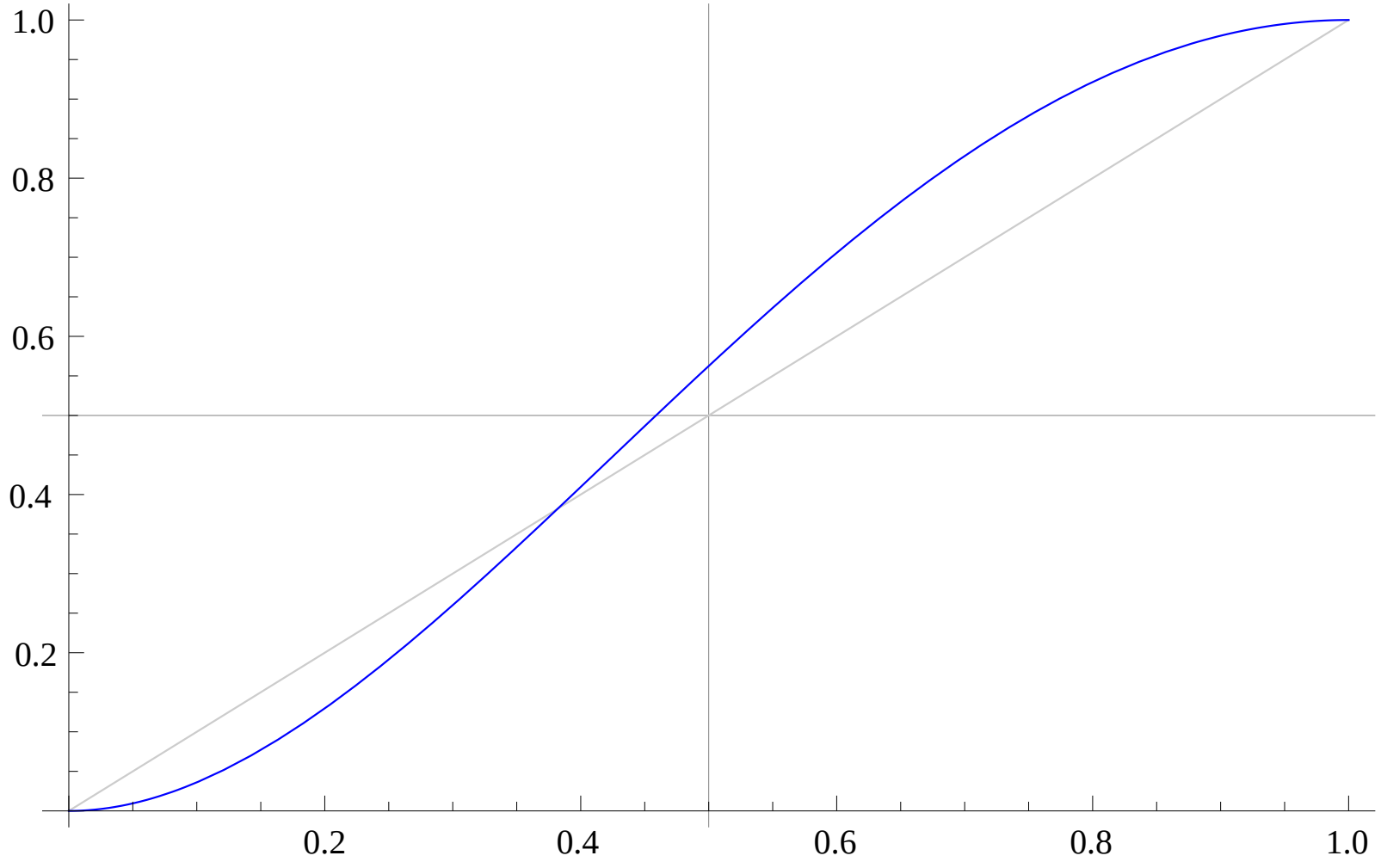
It follows that for input with bias $\Pr[x = 1] = p$ we have

$$\Pr[T_1(\boldsymbol{x}) = 1] = p^2(p-2)^2$$

Note that $p^2(p-2)^2[p \mapsto 1/2] = 9/16$.

In fact, T_1 increases the bias for $p \geq 1/2(3-\sqrt{5}) \approx 0.38$. This is plausible since both "and" and "or" are monotonic. See the next plot.

$p^2(p-2)^2$



A Bound

Lemma

$\mathsf{E}[S_k] \leq 3^k = n^{\log_4 3} \approx 0.79.$

Proof. Induction on k. $k = 1$ is OK.

Now consider the gate $T_{k-1} \vee T_{k-1}$. Given input bias p, the expected number of variables inspected in evaluating this gate is $2 - p$.

But $\Pr[R_k > 1/2]$ implies that this number is bounded by $3/2 \cdot 3^{k-1} = 1/2 \cdot 3^k$.

T_k has two such or-gates, done.

□

MAX 3-SAT

Here is another example of a circuit evaluation problem where randomization has interesting effects. Recall the classical $\mathbb{NP}$-complete decision problem

Problem:	**3-Satisfiability**
Instance:	A Boolean formula Φ in 3-CNF.
Question:	Is Φ satisfiable?

Thus,

$$\Phi = \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_m$$

where

$$\Phi_i = z_{i,1} \vee z_{i,2} \vee z_{i,3}$$

and each $z_{i,j}$ is a literal: a variable or a negated variable (from a set $V = \{x_1, \ldots, x_n\}$).

An Optimization Version

We can turn 3-SAT into an optimization problem by trying to determine a truth assignment $\alpha : V \to \mathbf{2}$ that maximizes

$$S(\alpha) = \#\text{ clauses satisfied by } \alpha$$

Of course, Φ is satisfiable iff $S(\alpha) = m$ for some α, so there is little hope to find a polynomial time algorithm for the optimization problem.

But failure is handled a bit more gracefully here: the formula may be unsatisfiable but $S(\alpha)$ may still be interesting.

Exercise

Figure out what $S(\alpha) = m - 1$ means for the instances in the hardness proof.

A Randomized Algorithm

Here is a trivial randomized algorithm to tackle the optimization problem: pick the truth assignment α at random (i.e., toss a fair coin to determine $\alpha(x)$ for each $x \in V$.)

This strategy totally ignores Φ and is really fairly idiotic, but note the following. Assume the clauses all have 3 distinct literals.

Lemma

$ExpS = 7/8 \cdot m.$

Proof. There is exactly way α can fail to satisfy a particular clause Φ_i; which happens with probability $1/8$.

Writing S as a sum of indicator variables we get our claim. □

A Weird Consequence

But then there is at least one truth assignment α such that $S(\alpha) \geq 7/8\, m$.

This is rather counterintuitive. One would think that it is possible to construct the clauses in a way so they contradict each other massively and only a small fraction can be satisfied. Not so.

What if we wanted to construct such an α?

We repeat the random construction until a 7/8-assignment pops up.

Then E[#repetitions] is geometrically distributed and we need to determine

$$p = \text{Pr[random assgn. is 7/8]}$$

Calculating p

Let

$$p_i = \text{Pr}[\text{exactly } i \text{ clauses are satisfied}].$$

So $7/8m = \text{E}[S] = \sum_{i=1}^{m} i\, p_i$. Define ℓ to be the largest integer $< 7/8m$. Splitting the sum at ℓ yields

$$\begin{aligned} 7/8\, m &= \sum_{i \le \ell} i\, p_i + \sum_{i > \ell} i\, p_i \\ &\le \sum_{i \le \ell} \ell\, p_i + \sum_{i > \ell} m\, p_i \\ &= \ell(1-p) + mp \qquad\qquad \le \ell + mp \end{aligned}$$

Calculating p

But then

$$mp \geq 7/8m - \ell \geq 1/8$$

and hence $p \geq 1/(8m)$.

So we have to repeat the random choice only $8m$ times, on average.

- Algorithms and Probability
- Circuit Evaluation
3. Yao's Minimax Principle
- Order Statistics
- More Randomized Algorithms

Yao's Minimax Principle

One can use understanding of the performance of deterministic algorithms to obtain lower bounds on the performance of probabilistic algorithms. To make this work, focus on Las Vegas algorithms: the answer is always correct, but the running time may be bad, with small probability.

Given some input I, a Las Vegas algorithm A makes a sequence of random choices during execution. We can think of these choices as represented by a choice sequence $C \in \mathbf{2}^\star$.

Given I and C, the algorithm behaves in an entirely deterministic fashion: $A(I; C)$.

Inputs and Algorithms

Fix some input size n once and for all (unless you love useless subscripts).

$\mathcal{I}$ = collection of all inputs of size n

$\mathcal{A}$ = collection of all deterministic algorithms for $\mathcal{I}$

It is clear that $\mathcal{I}$ is finite, but it requires a fairly delicate definition of "algorithm" to make sure that $\mathcal{A}$ is also finite.

Look up "algorithm" and "computable function" on Wikipedia, and you will see the problem.

LV as a Distribution

We can think of a Las Vegas algorithm A as a probability distribution on $\mathcal{A}$: with some probability the algorithm executes one of the deterministic algorithms in $\mathcal{A}$.

This works both ways: given a probability distribution on $\mathcal{A}$ we can think of it as a Las Vegas algorithm (though this is not the way algorithm design works typically).

In the following, suppose we two probability distributions: σ for $\mathcal{A}$ and τ for $\mathcal{I}$.

We'll indicate selections according to these distributions by subscripts.

Yao's Theorem

$$\min_{A \in \mathcal{A}} \mathsf{E}[T_A(I_\tau)] \leq \max_{I \in \mathcal{I}} \mathsf{E}[T_{A_\sigma}(I)]$$

Thus, the average case (wrt τ) running time of the best deterministic algorithm is a lower bound for the expected (wrt σ) running time of the corresponding Las Vegas algorithm on the worst input.

The proof is by computation: show that $\sum_{(A,I)} \mathsf{Pr}[A]\mathsf{Pr}[I]T_A(I)$ separates the two values. Note that we are not assuming independence!

Application: Minimax Circuits

There is a natural Las Vegas algorithm to evaluate T_k: at every node in the tree, pick a subtree at random, evaluate it and then determine whether the other tree also needs to be evaluated.

From what we have seen, this algorithm will evaluate $O(n^{0.79})$ variables on average on any input $I \in \mathbf{2}^{4^k}$.

To obtain a lower bound we need to understand $\mathcal{A}$, the class of all deterministic algorithms.

We claim that the performance of any deterministic algorithm can be matched or beaten by a top-down lazy algorithm, so we only need to consider these. This is not obvious, think about the necessary argument.

Dirty Trick: Xor

A simple computation shows that

$$T_1(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2) \oplus (x_3 \oplus x_4)$$

So, we can think of T_k as a homogeneous xor-tree of depth $2k$.

If we provide input to an xor gate with bias p than the output has bias $(1-p)^2$.

The equation $(1-p)^2 = p$ has solution $p = 1/2(3-\sqrt{5})$ and is visible as a fixed point in the graph in a previous slide.

Cost in Deterministic Algorithm

Let S_d be the cost of evaluating a node at depth d in the xor tree by some top-down lazy method.

$$\begin{aligned}\mathsf{E}[S_d] &= p \cdot \mathsf{E}[S_{d-1}] + (1-p) \cdot 2\mathsf{E}[S_{d-1}] \\ &= (2-p)\mathsf{E}[S_{d-1}] \\ &= 1/2(1+\sqrt{5})\mathsf{E}[S_{d-1}]\end{aligned}$$

It follows that $\mathsf{E}[S_{2k}] \approx n^{0.69}$, so no Las Vegas algorithm can do better than that in the worst case (i.e., worst input).

- Algorithms and Probability
- Circuit Evaluation
- Yao's Minimax Principle

4. Order Statistics

- More Randomized Algorithms

Rank and Order

Let $\mathcal{U}$ be some ordered universe such as the integers, rationals, strings, and so forth.

It is easy to set that for any set $A \subseteq \mathcal{U}$ of size n there is a unique order isomorphism

$$[n] \longleftrightarrow A$$

$\rightarrow$: $\mathsf{ord}(k, A)$

$\leftarrow$: $\mathsf{rk}(a, A)$

Note that $\mathsf{ord}(k, A)$ is trivial to compute if A is sorted.

Computation of $\mathsf{rk}(a, A)$ requires to determine the cardinality of $A^{\leq a} = \{\, z \in A \mid z \leq a \,\}$ (which is easy if A is a sorted array and we have a pointer to a).

Randomized Quicksort

Recall randomized quicksort. For simplicity assume elements in A are unique.

- Pick a pivot $s \in A$ uniformly at random.
- Partition into $A^{<s}$, s, $A^{>s}$.
- Recursively sort $A^{<s}$ and $A^{>s}$.

Here A is assumed to be given as an array. Partitioning takes linear time (though is not so easy to implement in the presence of duplicates).

Running Time

Let X be the random variable: size of $A^{<s}$. Then

$$p_i = \mathsf{Pr}[X = i] = 1/n$$

where $i = 0, \ldots, n-1$, $n = |A|$.

Ignoring multiplicative constants we get

$$t(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ \sum_{i<n} p_i(t(i) + t(n-i-1)) + n & \text{otherwise.} \end{cases}$$

Simplifying

$$
\begin{aligned}
t(n) &= 1/n \sum_{i<n} \big(t(i) + t(n-i-1)\big) + n \\
&= 2/n \sum_{i<n} t(i) + n \\
n \cdot t(n) &= 2 \sum_{i<n} t(i) + n^2 \\
(n+1) \cdot t(n+1) &= 2 \sum_{i \leq n} t(i) + (n+1)^2 \\
t(n+1) &= (n+2)/(n+1) \cdot t(n) + (2n+1)/(n+1)
\end{aligned}
$$

which comes down to (really $\leq$)

$$
t(n) = \frac{n+1}{n} \cdot t(n-1) + 2.
$$

Solving

$t(n) = n + 1/n \cdot t(n-1) + 2$ can be handled in to ways:

- Unfold the equation a few levels and observe the pattern.
- Solve the homogeneous equation $h(n) = n + 1/n \cdot h(n-1)$: $h(n) = n + 1$. Then construct t from h – see any basic text on recurrence equations.

Either way, we find

$$t(n) = (n+1)/2 + 2(n+1) \sum_{i=3}^{n+1} 1/i = \Theta(n \log n)$$

Random versus Deterministic Pivots

Random pivot:

$$\begin{aligned} \mathsf{Pr}[X = k] &= 1/n \qquad k = 0, \ldots, n-1 \\ \mathsf{E}[X] &= (n-1)/2 \\ \mathsf{V}[X] &= (n^2-1)/12 \end{aligned}$$

Median of three:

$$\mathsf{Pr}[X = k] = \frac{6k(n-k-1)}{} k = 1, \ldots, n-2$$

$$\begin{aligned} \mathsf{E}[X] &= (n-1)/2 \\ \mathsf{V}[X] &= ((n-1)^2 - 4)/20 \end{aligned}$$

Selection versus Sorting

While selection seems somewhat easier than sorting, it is not clear that one can avoid something like $O(n \log n)$ in the process of computing ord(k, A).

The following result was surprising.

Theorem (Blum, Floyd, Pratt, Rivest, Tarjan, 1973)

Selection can be handled in linear time.

The algorithm is a perfectly deterministic divide-and-conquer approach. Alas, the constants are bad.

Alternatively, we can use a randomized algorithm to find the kth element quickly, on average.

Probabilistic Selection

Given a collection A of cardinality n, a rank $1 \leq k \leq n$. Here is a recursive selection algorithm:

- Permute A in random order, yielding $a_1, a_2, \ldots, a_n$; set $B =$ nil.
- Pick a pivot $s \in A$ at random and compute $A^{<s}$ and $A^{>s}$. Let $m = |A^{<s}|$.
- If $k = m$ return s.
- If $k < m$ return ord$(k, A^{<s})$.
- If $k > m$ return ord$(k - m + 1, A^{>s})$.

Running Time

Correctness is obvious, for the running time analysis divide $[n]$ into bins of exponentially decreasing size: bin k has the form

$$B_k = [n \cdot (3/4)^k, n \cdot (3/4)^{k+1}]$$

where we ignore the necessary ceilings and floors, as well as overlap.

Note that with probability $1/2$ the cardinality of the selection set will move (at least) to the next bin in each round. But then it takes 2 steps on average to get (at least) to the next bin.

Hence the expected number of rounds is logarithmic and the total running time therefore linear.

- Algorithms and Probability
- Circuit Evaluation
- Yao's Minimax Principle
- Order Statistics

5 More Randomized Algorithms

Randomized Incremental Algorithms

Occasionally the construction of a data structure can be simplified significantly if on assumes the input is sufficiently random: one can then build the data structure in a very brute-force, step-by-step manner that requires no complicated ideas and is fast on average.

For example, suppose we wish to construct a sorted list B from a given list A.

- Permute A in random order, yielding $a_1, a_2, \ldots, a_n$; set $B =$ nil.
- for $k = 1, \ldots, n$: insert a_k into B, in the proper place.

Quoi?

This looks like insertion sort, wo why bother?

Because it isn't: we are going to maintain an additional data structure, a table that determines for each $x \in A - B$ which interval I defined by B element x belongs to. Moreover, for each interval I the table provides a list of all the elements in the interval.

Given the table, the insert step plus maintenance of the table can be handled in $O(|I|)$ steps.

So we need to find the expected value of the sum of the lengths of the intervals that we insert into.

A Trick: Going Backwards

Here is a trick that sometimes makes the argument a bit easier: run the algorithm backwards.

Here, going backwards in stage k means this: we randomly pick one of the k elements in B and remove it. Since the points in B are random, we should expect intervals of size n/k.

But then the total number of steps will about $nH_n = \Theta(n \log n)$, the best a comparison based sorting algorithm can do.

Alas, in practice, maintaining the table is cumbersome, so in the Real World this method is not competitive.

Convex Hulls

A set $A \subseteq \mathbb{R}^2$ is convex iff for all $x, y \in A$, the line segment $[x, y]$ is contained in A.

Note that $[x, y] = \{\, \lambda x + (1 - \lambda)y \mid 0 \leq \lambda \leq 1 \,\}$.

Given an arbitrary set A, the convex hull of A is defined to be the least convex set containing A:

$$\mathsf{ch}(A) = \bigcap \{\, C \mid A \subseteq C, C \text{ convex} \,\}.$$

This is a hull operation:

- $A \subseteq \mathsf{ch}(A)$.
- $\mathsf{ch}(\mathsf{ch}(A)) = \mathsf{ch}(A)$.

Better Description

Note that the definition as stated is impredicative and hence not too useful (ch(A) is one of the sets on the right hand side). Here is a better one:

$$\text{ch}(A) = \{ \sum \lambda_i a_i \mid \sum \lambda_i = 1, 0 \leq \lambda_i, a_i \in A \}$$

The $\sum \lambda_i a_i$ are called convex combinations.

In particular when A is finite, say $A = \{a_1, \ldots, a_n\}$, we can obtain the hull as

$$\text{ch}(A) = \{ \sum \lambda_i a_i \mid \sum \lambda_i = 1, 0 \leq \lambda_i \}$$

Extremal Points

Some of the a_i can be expressed as convex combinations of others, so the problem comes down to identifying $B \subseteq A$ such that $\mathsf{ch}(B) = \mathsf{ch}(A)$ but no proper subset works.

Hence a reasonable output format for the convex hull is to return a list

$$b_1, b_2, \ldots, b_m$$

of extremal points, obtained by traversing them in clockwise order, starting at the "top-left" point.

Lower Bound

As a consequence of our output convention, we get a lower bound: we can use the convex hull to sort. To see why, suppose we have integral or rational numbers $x_1, \ldots, x_n$.

Define points $a_i = (x_i, x_i^2)$ on the parabola $y = x^2$.

Since the parabola is convex one can read the sorted list off the convex hull of A.

We will now match this bound with a randomized incremental algorithm to construct the hull.

For simplicity assume that A contains no collinear points.

Randomized Incremental Algorithm

- Permute A in random order, yielding $a_1, a_2, \ldots, a_n$;
- Let $B = (a_1, a_2, a_3)$, let c be the centroid of this triangle.
- for $k = 4, \ldots, n$: insert a_k into B:
  - if $a_k \in \mathsf{ch}(B_{k-1})$ do nothing
  - otherwise modify B_{k-1} to include a_k.

As before, we will need to maintain additional information: for each point $a \in A - B$ the edge of the convex hull of B that intersects the line segment $[c, a]$.

In the opposite direction, we need for each edge a list of all the corresponding points.

Analysis

Updating B may require the removal of $O(n)$ points from B, but the total number of removals is bounded by $2n$: we insert at most $2n$ points and we can charge for removal at the moment of insertion.

So the critical part is the update operation on the edge-points table: we need to process all the points currently associated with the edge that is being removed from the boundary of B_{k-1}.

Using the backward trick, the argument is precisely the same as for the sorting algorithm from above.

More Randomized Selection

Here is a randomized algorithm for selection that uses a few magic numbers. The numbers make sense only when one performs a probabilistic analysis of the algorithm.

Convention: We will systematically ignore ceilings and floors and pretend that various numbers such as $\sqrt{n}$ are integral.

We are given a set $A \subseteq \mathcal{U}$ of n elements and we would like to determine $t = \mathsf{ord}(k, A)$.

To this end, the algorithm selects a "small" subset B of A and works with B. Actually, we sample A with replacement.

Batten down the hatches.

Crazy Selection (Las Vegas)

1. Sample A with replacement $n^{3/4}$ times to produce B.
2. Sort B.
3. Let $\kappa = k/n^{1/4}$, $\kappa^- = \max(\kappa - \sqrt{n}, 1)$, $\kappa^+ = \min(\kappa + \sqrt{n}, n^{3/4})$, $b^\pm = \mathsf{ord}(\kappa^\pm, B)$.
4. Compute $r^\pm = \mathsf{rk}(b^\pm, A)$ – note the A.
5. Let
$$A_0 = \begin{cases} \{\, x \in A \mid x \le b^+ \,\} & \text{if } k < n^{1/4}, \\ \{\, x \in A \mid x \ge b^- \,\} & \text{if } k > n - n^{1/4}, \\ \{\, x \in A \mid b^- \le x \le b^+ \,\} & \text{otherwise.} \end{cases}$$
6. if $t \notin A_0$ or $|A_0| > 4n^{3/4}$ return to step 1.
7. Sort A_0 and return $\mathsf{ord}(k - r^- + 1, A_0)$.

Comments

- Think of $n = 10^8$ so that $n^{3/8} = 10^6$ and $\kappa = k/100$.
- It is easier to pretend that B is a subset of A cardinality $n^{3/4}$. Alas, picking a subset of this size would make the algorithm more clumsy to implement and harder to analyze.
- In an ideal scenario, the elements in B would be equidistant; in that case we only would need to consider the interval spanned by the immediate neighbors of $\mathrm{ord}(\kappa, B)$ in B. By going out to $\sqrt{n}$ we hope to compensate for the fact that B is not regularly placed.
- Let's count comparisons. The only part that is expensive is step (4), the total damage is $2n + o(n)$.
- The test in (6) is not impossible: we use the order isomorphism and check $r^- \leq k \leq r^+$ instead.

Analysis

Lemma

The Crazy Selection algorithm terminates after one round with probability $1 - O(n^{-1/4})$.

Proof. Unfortunately, there are several cases to consider. For simplicity, we deal only with

$$A_0 = \{\, x \in A \mid b^- \leq x \leq b^+ \,\}$$

and show that $t \notin A_0$ is unlikely. $t \notin A_0$ means $t < b^-$ or $t > b^+$. In the first case we must have

$$\#(x \in B \;\big|\; x \leq t) < \kappa^-$$

and in the other case

$$\#(x \in B \;\big|\; x \leq t) > \kappa^+.$$

Analysis, cont'd.

This suggests to consider to random variable

$$X = \#(x \in B \mid x \leq t)$$

which can be written as an indicator variable sum $X = \sum_{i=1}^{n^{3/4}} X_i$ where $X_i = 1$ iff the ith element in B is $\leq t$.

Note that we sample A with replacement and "ith element" means in the order of selection; X is really the number of samples below t (but for intuition think of it as cardinality).

It follows that $\Pr[X_i = 1] = k/n$.

Analysis, cont'd.

Clearly the X_i are Bernoulli, so we can calculate stats for X as follows:

$$\begin{aligned}
\mathsf{E}[X] &= k/n \cdot n^{3/4} = kn^{-1/4} = \kappa \\
\mathsf{V}[X] &= n^{3/4} \cdot k/n \cdot (1 - k/n) \leq 1/4 \cdot n^{-1/4} \\
\sigma &\leq 1/2\, n^{3/8}
\end{aligned}$$

The bound on $\mathsf{V}[X]$ follows from considering the parabola $x(1-x)$.

By Chebyshev,

$$\mathsf{Pr}[|X - \kappa| \geq \sqrt{n}] \leq \mathsf{Pr}[|X - \kappa| \geq 2n^{1/8}\sigma] = O(n^{-1/4})$$

Analysis, cont'd.

It follows that $\Pr[t < b^-] = O(n^{-1/4})$.

Essentially the same argument shows that $\Pr[b^+ < t] = O(n^{-1/4})$.

But the probability of the union of the two failure modes is bounded by the sum of the respective probabilities, which is still $O(n^{-1/4})$.

□

Note that the bound $O(n^{-1/4})$ is not overwhelming; we have not even made an attempt to estimate the constants.

We certainly would not want to use a recursive version of the algorithm.