EECS 477: Introduction to algorithms.
Lecture 4
TTh 8:30-10am

Prof. Igor Guskov
guskov@eecs.umich.edu
office: 126 ATL building (AI lab)

September 12, 2002
Lecture outline

• Computational models

• Problems and instances

• Sorting

• Amortized analysis: binary counter
• **Lemma:** \( \text{GCD}(m, n) = \text{GCD}(m \% n, n) \) if \( m \% n \neq 0 \)

• Indeed, we have \( m = nt + r \) (\( r := m \% n \)) since every common divisor of \( m \) and \( n \) divides \( r \), and every common divisor of \( n \) and \( r \) divides \( m \).

It follows then that \( \text{GCD}(m, n) \leq \text{GCD}(r, n) \) and \( \text{GCD}(m, n) \geq \text{GCD}(r, n) \) which proves the above claim.

• From the lemma it follows that the Euclid algorithm is correct.
Computational models

- Elementary operations: execution time is bounded by a constant that does not depend on input values. Actual seconds per operation may be disregarded. Selection of elementary operations is hardware dependent.

- Arithmetic operations: +, -, *, etcetera

  Yes: if integers have bounded number of bits

  No: otherwise (unbounded)

- Function calls, memory accesses, e.g. a[i]?
Computational models II

• Computational model is determined by: data representation and storage mechanisms, available elementary operations. Examples: logic circuits, deterministic finite automata, push-down automata, Turing machines, C/C++ programs
External memory: two-level disk model – count I/O operations

• Computational models originate in technologies, physics, biology, etcetera

• Parallel and distributed computing, optical and DNA computing, analog computing, quantum computing
Elementary operations

- Elementary = execution time bounded by a constant

  Example: an algorithm performs $A$ additions, $M$ multiplications, and $S$ assignments; and each addition take no longer than $t_A$, each multiplication no longer than $t_M$, and each assignment no longer than $t_S$. Then the total execution time is bounded by

  $$ T \leq At_A + Mt_M + St_S \leq \max(t_A, t_M, t_S)(A + M + S) $$

  so the number of elementary operations is a decent measure of performance.

- Of course, we need to be careful with what operations are elementary. For instance, prime testing and long integer multiplications are not elementary.
Problems vs instances

• Problem: in the *functions domain*

• Instance: one possible argument – function input

• For instance, C(N,k) vs C(20, 3)

• Instances can be different: best and worst case, average (ex: sorting insertion (sensitive) vs selection(insensitive))

• Instances that require average resources are called *average case instances*
Insertion sort: sensitive

- // grows sorted range
  void insertion_sort(vector<float>& a) {
    for(i=1; i<a.size(); ++i) {
      float x = a[i];
      int j = i - 1;
      while(j >= 0 && a[j] > x) {
        a[j + 1] = a[j];
        --j;
      }
      a[j] = x;
    }
  }
Insertion sort II

- The best case: sorted sequence (body of the while loop never executes). Results in linear number of operations

- The worst case: reverse (body of the while loop executes \( i \) times): \[ 1 + 2 + \ldots + (n - 1) = (n - 1)n/2 \]

- Average case: still quadratic – average number of times the body of the while loop is performed is roughly \( i/2 \)
Selection sort: insensitive

- // puts minimal element first and moves right
  void selection_sort(vector<float>& a) {
      for(i=0; i<a.size()-1; ++i) {
          int minj = i; float minx = a[i];
          for(int j=i+1; j<a.size(); ++j) {
              if(a[j]<minx) {
                  minj = j; minx = a[j];
              }
          }
          a[minj] = a[i];
          a[i] = minx;
      }
  }

Selection sort II

- The best case: quadratic number of if-comparisons

- The worst case: quadratic number of if-comparisons

- Average case: (what else but) quadratic number of if-comparisons

- this algorithm’s performance is insensitive to the input
Quicksort

- // pivot and recurse
  void quick_sort(iterator iBegin, iterator iEnd) {
    iterator iPivot = pivot(iBegin, iEnd); // split
    assert(iPivot!=iEnd);
    quick_sort(iBegin, iPivot);
    ++iPivot;
    quick_sort(iPivot, iEnd);
  }

- Average case performance is $n \log n$, worst case in quadratic
Efficiency

• Is it a big deal?

• Three algorithms: \( A_0 \) takes \( C_0 \log n \) time, \( A_1 \) takes \( C_1 n^d \) time, and \( A_2 \) takes \( C_2 2^n \) time on instance of size \( n \). Let’s say all of them run one day on instance of size \( N \).

• You bought a new machine that runs twice faster: how big of an instance could you process now?

• Generally, \( t = Cf(n) \) so that \( n = f^{-1}(t/C) \) and \( t/C \) doubles.
Efficiency II

- $A_0$: we had $2^{t/C_0} = N$, now we have $2^{2^{t/C_0}} = N^2$

- $A_1$: we had $\sqrt{t/C_0} = N$, now we have $\sqrt{2^{t/C_0}} = \sqrt{2}N$

- $A_2$: we had $\log(t/C_0) = N$, now we have $\log(2^{t/C_0}) = N + 1$

So polynomial algorithms are okay, exponential are not good.
Amortized analysis: binary counter

class CounterT {
    void Increment() {
        j = 0;
        do {
            b[j] = 1 - b[j];
            if(b[j]==1)
                break;
            ++j;
        } while( j<M );
    }

    // the counter’s value is b[0]+2*b[1]+...+2^[M-1]*b[M-1]
    bit b[M];
}

Amortized analysis: binary counter II

• a single call to increment changes as many bits as there are trailing ones in the counter binary representation

• e.g.: Increment(0101111) = 0110000

• An easy upper bound on one call is then constant times $M$.

• What if we call $bc.Increment()$ repeatedly, $N$ times?
Potential function

- Let $\phi(bc)$ be the number of bits equal to one in $bc$

- Define the amortized cost of execution $t_i^a = t_i + \phi_i - \phi_{i-1}$. This allows to measure how “dirty” the current state is.

- Call on an even integer has amortized cost $1 + 1 = 2$ (one bit changed, and one more “1” added)
  Call on “11....1” has amortized cost of $M - M = 0$
  Otherwise, a call changes $k$ bits and $k - 1$ of them from one to zero and one from zero to one resulting in $k - (k - 2) = 2$ amortized cost.
Amortized analysis: binary counter III

- Any single call has amortized cost less than 2

- Overall time is \( \sum_{i=1}^{N} t^a_i = \phi_N - \phi_0 + \sum_{i=1}^{N} t_i \)

- Therefore, \( N \) operation take \( 2N \) time plus total change in the dirtiness measure which is bounded by \( M \).

- Hence, amortized const on a single cost to increment a counter is \( 2 + M/N \) and when \( N \) is big that is constant!