MCS-375: Algorithms: Analysis and Design Handout #A4
San Skulrattanakulchai
Gustavus Adolphus College Sep 12, 2014

"Exponential" Algorithms

Fix a problem having input size n. The term "algorithm" in this handout means an algorithm for solving this problem. Our convention is to assume all running time are worst-case, unless we state otherwise.

A *good* algorithm runs in polynomial time $O(n^k)$ for some $k > 0$. A problem that admits a good algorithm is said to be *tractable*; it is said to be *intractable* otherwise.

A *superpolynomial* time algorithm runs in time $\omega(p)$ for any polynomial function p.

An *exponential* time algorithm runs in time $\Omega(a^n)$ for some constant $a > 1$.

Notes

(i) Any polynomial function is better than any exponential function.

Theorem. *n^a is $O(b^n)$ for any constants a, b, where $a > 0$ and $b > 1$.*

Proof. ... □

(ii) A super-polynomial time algorithm can be better than an exponential-time algorithm.

Exercise Find a function that is super-polynomial but not exponential.

(iii) The running time of some algorithm is worse than any exponential. For example, $f(n) = n!$ is not $O(g)$ for any exponential function g.

Theorem. *a^n is $O(n!)$ for any constant a.*

Proof. ... □

(iv) It's customary to refer to the superpolynomial functions as exponential. This sloppy language is widespread, probably as widespred as the incorrect usage of "its" for "it's" (and vice versa) in English.

Two "exponential-time" algorithms

The Subset Sum Problem (SSUM) Given a set S of n positive integers and a target integer t, find out if there exists a subset of S whose sum is exactly t.
Example: Let $S = \{1, 2, 6\}$.
If target $t = 3$, then answer is YES since numbers in $\{1, 2\}$ sum to 3.
If target $t = 4$, then answer is NO since no subset of S sums to 4.

Here is a naive algorithm for SSUM.

```
for each subset T of S do
    if the elements of T sum to t then
        return YES
return NO
```

The algorithm can check 2^n possible solutions and so it has an exponential time bound. Here is an implementation of the above high-level algorithm.

```
procedure main {
    S ← input array of positive integers
    t ← input target number
    n ← S.length
    solution ← NO
    check(1, 0)
    return solution
}
procedure check(i, sum) {
    if i ≤ n then {
        check(i + 1, sum)    /* don't include S[i] in the set */
        check(i + 1, sum + S[i])    /* include S[i] in the set */
    } else if sum = t then
        solution ← YES
}
```

The Travelling Salesman Problem (TSP) Given n cities and the distances between all ordered pairs of distinct cities, find a shortest route that starts at city 1, visits each city, and returns to city 1.

Here is a "naive algorithm" for TSP.

```
/* algorithm sets min_dist to length of a shortest salesman tour */
procedure main {
    min_dist ← ∞
    for each permutation π of the integers 2, ..., n do {
        d ← length of the route starting at city 1,
            visiting cities 2, ..., n in the order given by π,
            and returning to city 1;
        min_dist ← min{min_dist, d}
    }
    return min_dist
}
```

This high-level algorithm checks $(n-1)!$ possible solutions, each in time $\Omega(n)$. So its running time is $\Omega(n!)$.

The next low-level algorithm shows that this high-level algorithm can be implemented so that the running time is $O(n!)$.

Here is an implementation of the above high-level algorithm. Assume the distances are given as the 2-d array `dist`$[1..n, 1..n]$. We use `pi`$[1..n]$ to keep the permutations.

```
procedure main {
    min_dist ← ∞
    for j ← 1 to n do
        pi[j] ← j
    perm(2)
    return min_dist
}
procedure perm(i) {
    if i = n then {
        d ← dist[pi[n], 1]
        for j ← 1 to n − 1 do
            d ← d + dist[pi[j], pi[j + 1]]
        if d < min_dist then
            min_dist ← d
    } else {
        for j ← i to n do {
            swap(pi[i], pi[j])
            perm(i + 1)
            swap(pi[i], pi[j])
        }
    }
}
```