COMP 6651

Algorithm Design Techniques

Week 2

Reductions. Recurrences. Divide and Conquer

(some material is taken from web or other various sources with permission)

Algorithm Analysis

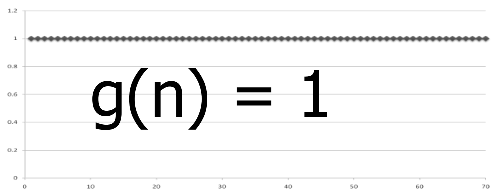
Asymptotic Analysis

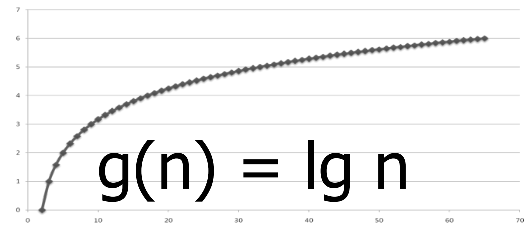
Questions we want answered:

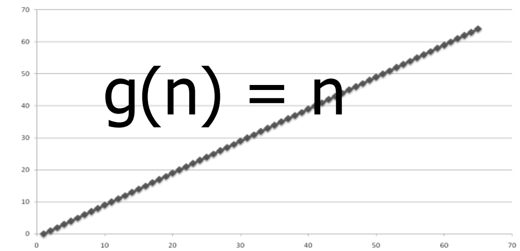
1. What algorithm is faster in the limit (as N (the size of input) grows towards infinity)
2. How practical is the algorithm (what category)
3. Is it possible to do better

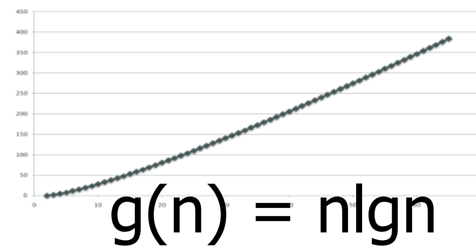
Algorithm Analysis

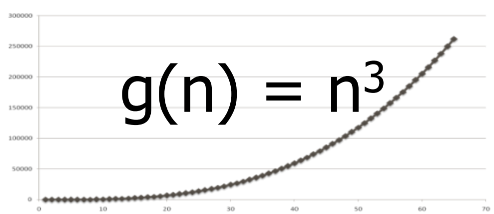
1. How practical is the algorithm (what category)

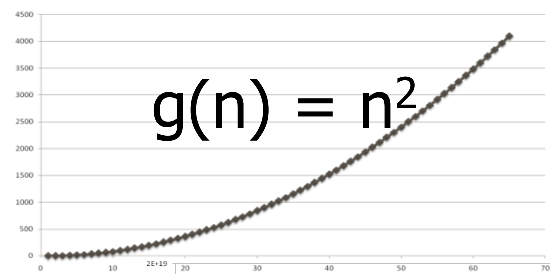


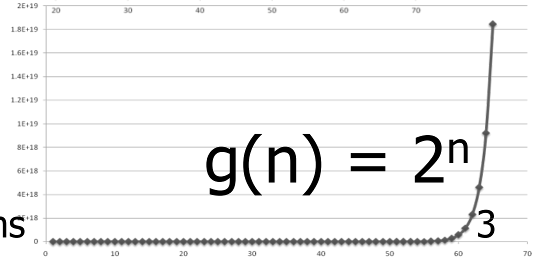








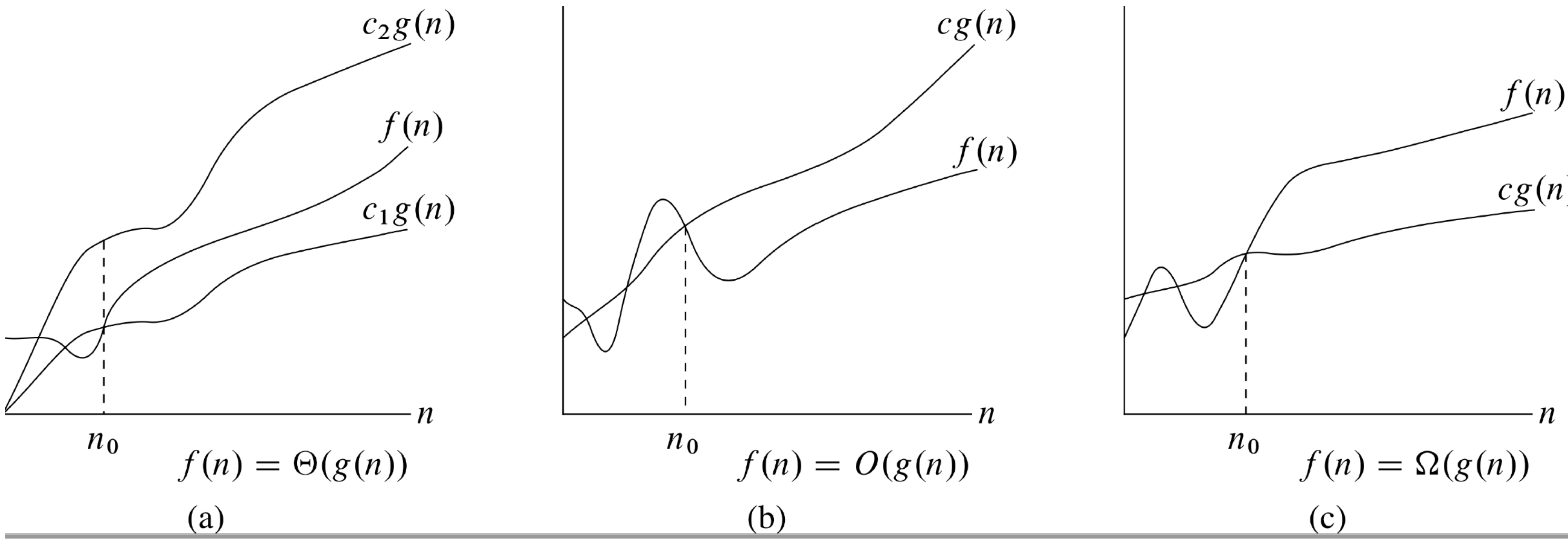




Algorithm Efficiency

- $O(g(n)) = \{f(n) | \exists\, c > 0, n_o > 0\ s.t.\ 0 \leq f(n) \leq c\big(g(n)\big) \forall\, n > n_0\}$
- OK, but what is it? Looks like ancient greek
- O()→ is a SET of functions
  - Domain: space of functions
  - Input is a function
  - Co-domain/ output: set of function
- For example:
  - What is O(n) ?
  - (…. on the board ….)

Algorithm Analysis



Algorithm Efficiency

- $O(g(n)) = \{f(n) | \exists\, c > 0, n_o > 0\ s.t.\ 0 \leq f(n) \leq c \cdot g(n). \forall\, n > n_0\}$
- Difficult to compute all functions belonging to this class
- More useful to ask if a function f(n) belongs to O(g(n))
- Examples:
  - Is $\mathrm{f}(n) = 2n + 1 \in O(n)$? Abuse of notation we say
    - IS $\mathrm{f}(n)\ \ O(n)$?
  - Is $\mathrm{f}(n) = 2n + \log(n) \in O(n)$?
  - Is $\mathrm{f}(n) = n^2 + n \in O(n)$?
  - Is $\mathrm{f}(n) = 2^{n+7} + n \in O(2^n)$?
  - Is $\mathrm{f}(n) = n + 1 \in O(nlogn)$?
  - Is $O(n) == O(3n + 15)$?
  - Is $O(n) == O(an + b), a \neq 0$?

Algorithm Efficiency

- $O(g(n)) = \{f(n) | \exists\, c > 0, n_o > 0\ s.t.\ 0 \leq f(n) \leq c \cdot g(n).\ \forall\ n > n_0\}$
- Difficult to compute all functions belonging to this class
- More useful to ask if a function f(n) belongs to O(g(n))
- Examples:
  - Is $O(n) == O(n^2)$?
  - Is $\mathrm{f}(n) = n^2 + n \in \Omega(n)$?
  - Is $\mathrm{f}(n) = n^2 + n \in \Theta(n)$?

Algorithm Efficiency

$O(g(n)) = \{f(n) |\ \exists\ c > 0, n_o > 0\ s.t.\ 0 \leq f(n) \leq c \cdot g(n). \forall\ n > n_0\}$

o$(g(n)) = \{f(n) | \forall c > 0, n_o > 0\ s.t.\ 0 \leq f(n) \leq c \cdot g(n). \forall\ n > n_0\}$

$\Omega\ (g(n)) = \{f(n) |\ \exists\ c > 0, n_o > 0\ s.t.\ f(n) \geq c \cdot g(n) \geq 0. \forall\ n > n_0\}$

$\omega\ (g(n)) = \{f(n) | \forall c > 0, n_o > 0\ s.t.\ f(n) \geq c \cdot g(n) \geq 0. \forall\ n > n_0\}$

$\theta\ (g(n)) = \{f(n) |\ \exists\ c_2, c_2 > 0, n_o > 0\ s.t.\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n). \forall\ n > n_0\}$

- Notes:
  - O and Ω – opposites
    - O – upper bound
    - Ω – lower bound
    - θ is both (sandwich)

Algorithm Efficiency

- In this class we use "mostly" O, Ω and θ
- O and Ω – opposites
  - O – upper bound
  - Ω – lower bound
  - θ is both (sandwich)
- Why do we care about the lower bound?
- Gives a theoretical limit
- Stop looking at faster algorithms
- Useful in reductions

Algorithm Efficiency

- Classical Example: sorting
  - You can prove that is O(nlogn) by providing an algorithm that is O(nlogn)
  - For instance: merge sort – O(nlogn)
  - If you can prove that is also Ω(nlogn)
    - nlogn → optimal algorithm

  - Exercise: prove that $f = O(g)$ and $f = \Omega(g)$ iff f = θ(g)

Algorithm Efficiency Algorithm

```
/* n input size, n>0 */
/* v - array of integers of size n */
/* returns the smallest difference in the absolute
value between all pairs of elements of v */


int smallest = -1;
for(int i=0;i<n;++i)
    for(int j=i+1;j<n;++j){
        int d = abs(v[i]-v[j]);
        if(smallest < 0 || smallest > d)
            smallest = d;
    }
return smallest;
```

Algorithm Efficiency Algorithm

```
/* n input size, n>0 */
/* v - sorted array of integers of size n */
/* x - integer number */
/* return an index k whose value v[k]==x OR -1 otherwise */
int findElement(v, x){
    return findElementI(v, 0, n-1, x);
}
int findElementI(v, int i1, int i2, x){
    if(i1==i2){
        if(v[i1]==x){
            return i1;
        }
        return -1;
    }
    int middle = (i1+i2)/2;
    if(x<=v[middle]){
      return findElementI(v, i1, middle, x);
    } else {
      return findElementI(v, middle + 1, i2, x)
   }
```

Algorithm Efficiency Algorithm

```
for(int i=0;i<n;i=i+100)
    for(int j=0;j<n;j=j+100000){
      … (constant time)
  }
```

```
for(int i=0;i<n;i=i+100)
    for(int j=1;j<n;j=j*2){
      … (constant time)
  }
```

Recursion

- Process of defining a problem of solution in terms of simpler versions of itself

Recursion

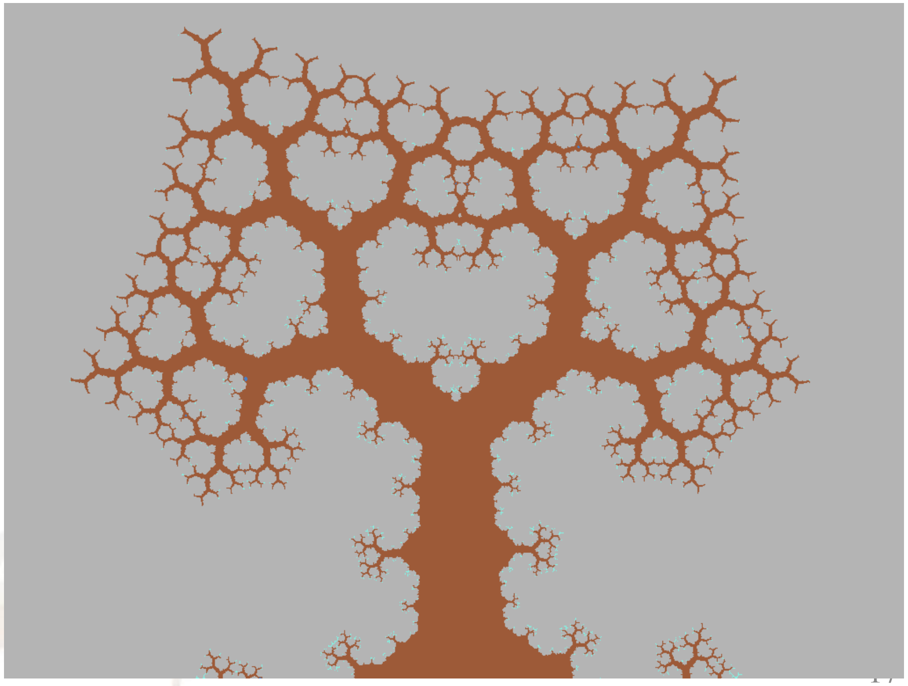
- Process of defining a problem of solution in terms of simpler versions of itself
- Examples:
  - n! = n*(n-1)!
  - $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ for all integers $n, k : 1 \leq k \leq n-1,$

Recursion

- Process of defining a problem of solution in terms of simpler versions of itself
- Examples:
  - n! = n*(n-1)!
  - $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ for all integers $n, k : 1 \leq k \leq n-1,$

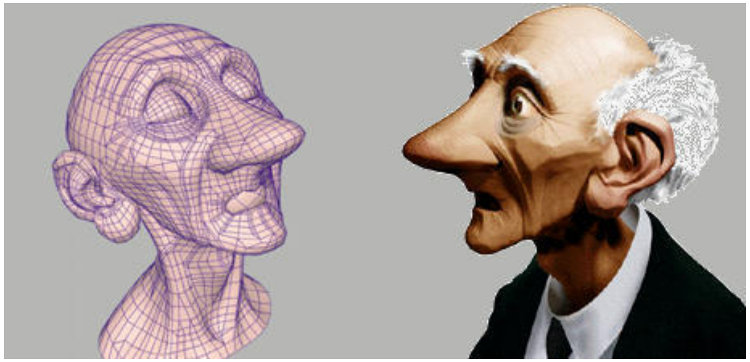
Recursion

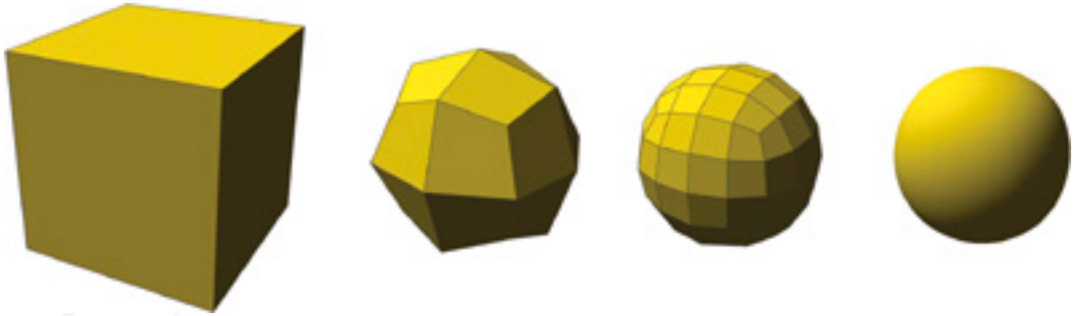
- Process of defining a problem of solution in terms of simpler versions of itself
- Examples:
  - Fractals



Recursion

- Process of defining a problem of solution in terms of simpler versions of itself
- Examples:
  - Subdivision surfaces:





Algorithm Efficiency Algorithm

```
int findmax(std::vector<int> v){
    int max = v[0];
    for(int i=1;i<v.size();++i)
        if(max<v[i])
            max = v[i];
}
```

Algorithm Efficiency Algorithm

```
int findmax(std::vector<int> v){
    return findmaxI(v, 0, v.size()-1);
}

int findmaxI(std::vector<int> v, int i1, int i2){
    if(i1==i2){
        return v[i1];
    } else {
        int middle = (i1 + i2)/2;
        int m1 = findmaxI(v, i1, middle);
        int m2 = findmaxI(v, middle+1, i2);
        if(m1<m2)
            return m2;
        else
            return m1;
    }
}
```

Divide and Conquer

- First major algorithmic paradigm
- Simple idea since the beginning of time
- Smaller problems are easier than large ones
- Break it down
- Divide/Conquer/Combine

Divide and Conquer

- Divide/Conquer/Combine

```
MERGE-SORT(A, p, r)
  if p < r                        // check for base case
      q = ⌊(p + r)/2⌋             // divide
      MERGE-SORT(A, p, q)         // conquer
      MERGE-SORT(A, q + 1, r)     // conquer
      MERGE(A, p, q, r)           // combine
```

Divide and Conquer

- Divide/Conquer/Combine

```
MERGE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i - 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

Divide and Conquer

- Is it always recursive
  - Well… yeah… sort-of
  - What is the problem with recursive calls?
  - Can we do it no—recursive?
  - Yes! –
  - Not only we can, but we should (especially when working in the industry)

Algorithm Efficiency Algorithm

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k : 1 \leq k \leq n-1,$$

```
/* compute n choose k */
int n_choose_k(int n, int k){

      if(k==0)
            return 1;

       if(n==k)
            return 1;

    int a = n_choose_k(n-1, k-1);
    int b = n_choose_k(n-1, k);

    return a + b;
}
```

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k : 1 \le k \le n-1,$$

Algorithm Efficiency Algorithm

```
/* compute n choose k */
int n_choose_k(int n, int k){

    Matrix M(n+1, k+1);

    // base case
    for(int i=0;i<n+1;++i)
        M(i, 0) = 1;

    for(int i=0;i<k+1;++i)
        M(i,i) = 1;


    for(int i=1;i<n+1;++i){
        int l = min(k+1, i);
        for(int j=1;j<l;++j)
            M(i,j) = M(i-1, j-1) + M(i-1, j);
    }

    return M(n,k);
}
```