

CSE 220: Handout 4
Algorithm Analysis

Puzzle solution

- Thm: If Bob dislikes n persons, then these n persons will realize that on n -th Friday
- Base Case: $n = 1$. Suppose Bob dislikes Alice. Alice knows that Bob does not dislike any other person. So when Bob makes his public admission, she realizes.
- Inductive Hypothesis: If Bob dislikes $n \leq k$ persons then they realize that on n -th Friday, and won't show up the next Friday.
- Inductive case: $n = k + 1$. Since everyone shows up on $(k + 1)$ -th Friday, everyone concludes that Bob dislikes more than k persons (by inductive hypothesis). Suppose Bob dislikes Alice. Alice knows only k other persons whom Bob dislikes, so she concludes...

Motivation for Algorithm Analysis

- Goal: Understanding of resource requirements of an algorithm
- Important computational resources: time and memory
- Running time analysis estimates the time required as a function of the input size
- Advantages:
 - Estimate growth rate as input grows
 - Guide to choose between alternatives

Sample analysis

```
largest(X) {  
    -- X is an array of n numbers  
1 int current = 0;  
2 for ( int i=0; i<n; i++)  
3     if ( X[i] > current)  
4         current = X[i];  
5 return current;  
}
```

- Input size: n (number of array elements)
- Goal: To count number of basic operations (e.g. comparison, assignment, increment) as a function of n
- Running time of *largest* is

$$T(n) = c \cdot n + d$$

for small constants c and d

Model of Computation

- Model of computation is an ordinary (sequential) computer
- Assumption: Basic operations take 1 time unit
- What are basic operations?
 - Arithmetic operations, comparisons, assignments etc.
 - Library routines such as *sort* should not be considered basic
 - Use common sense
- Formal models of computation, such as Turing machines, exist, and allow more precise analysis (CSE 262)

Big-Oh Notation

- A standard for expressing upper bounds
- The running time of *largest* is $O(n)$
that is, ignore constants c and d
- Definition: $T(n) = O(f(n))$ if there exist constants c and n_0 such that $T(n) \leq cf(n)$ for all $n \geq n_0$
- Intuitively, an algorithm A is $O(f(n))$ means that, if the input is of size n , the algorithm will stop after $f(n)$ time.

Examples

$$15n + 32 =$$

$$1324 =$$

$$2n^2 =$$

$$5n^2 + 10n + 6 =$$

$$n^2/100 =$$

$$c_1n^3 + c_2n^2 + c_3n + c_4 =$$

$$c2^n + dn^5 =$$

$$g(0) = 123, g(1) = 456, g(n) = 3n \text{ for } n \geq 2$$

Intuition for ignoring constants:

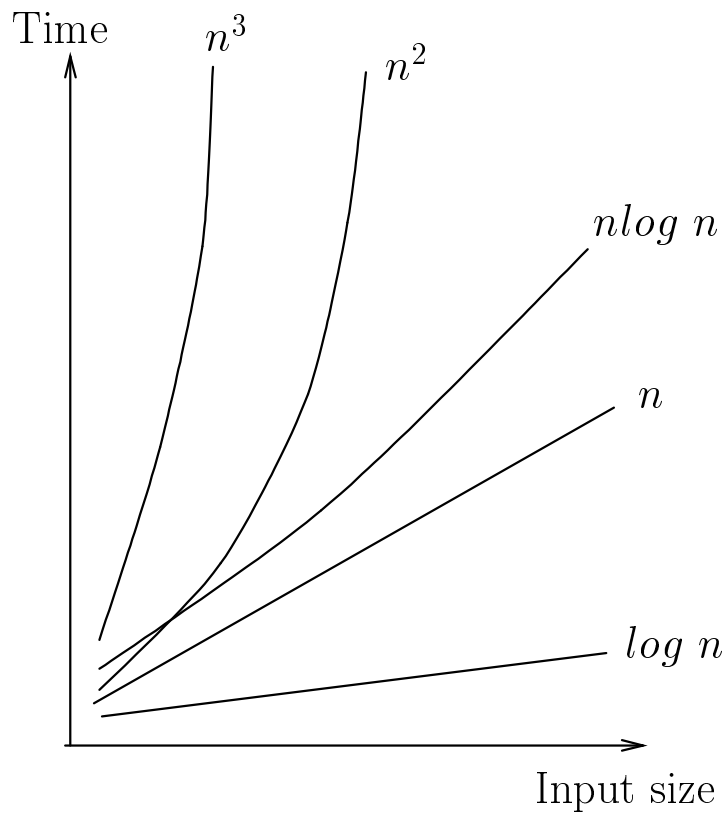
$2n^3$ will grow faster than $17n^2 + 400$

More on Big-Oh

- Definition does not require upper bound to be tight, though we would prefer as tight as possible
- Thus, it is correct to say that $5n^2+3$ is $O(n^3)$
- Typical bounds in increasing order:

$O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), O(2^n)$

Comparison of Growth Rates



Caution: Not to scale

Rules for Big-Oh

1. If $T = O(cf(n))$ for a constant c , then
 $T = O(f(n))$
2. If $T_1 = O(f(n))$ and $T_2 = O(g(n))$ then
 $T_1 + T_2 = O(\max(f(n), g(n)))$
3. If $T_1 = O(f(n))$ and $T_2 = O(g(n))$ then
 $T_1 * T_2 = O(f(n) * g(n))$
4. If f is a polynomial of degree k then
 $f(n) = O(n^k)$

Nested Loops

- Running time of a loop equals running time of code within the loop times the number of iterations
- Nested loops: analyze inside out

```
1 int k=0;
2 for ( int i=0; i<n; i++)
3     for ( int j=0; j<n; j++)
4         k++;
```

- Running time of lines 3-4: $O(n)$
- Running time of lines 1-4: $O(n^2)$
- Note: Running time grows with nesting rather than the length of the code

Sequential statements

For a sequence $S_1; S_2; \dots S_k$ of statements, running time is maximum of running times of individual statements

```
1 for ( int i=0; i<n; i++)
2     X[i] = 0;
3 for ( int i=0; i<n; i++)
4     for ( int j=0; j<n; j++)
5         X[i] += i+j;
```

Running time is:

Conditional statements

The running time of

```
if ( cond ) S1
else S2
```

is running time of *cond* plus the max of running times of *S1* and *S2*

```
int k=0;
for ( int i=0; i<n; i++)
    if even(i) {
        for ( int j=0; j<n; j++)
            k++; }
else
    k *= 2;
```

Running time:

More nested loops

```
1 int k=0;
2 for ( int i=0; i<n; i++)
3     for ( int j=i; j<n; j++)
4         k++;
```

- Running time of lines 3-4: $n - i$
- Running time of lines 1-4:

$$\sum_{i=0}^{n-1} (n - i) = n(n - 1)/2 = O(n^2)$$

More nested loops

```
1 int k=0;
2 for ( int i=1; i<n; i*=2)
3     for ( int j=1; j<n; j++)
4         k++;
```

- Running time of inner loop (lines 3–4): $O(n)$
- How many times is outer loop executed?
- In m -th iteration value of i is 2^{m-1}
- Suppose $2^{l-1} < i \leq 2^l$, then outer loop is executed l times
- Running time is $O(n \log n)$

Lower bounds

- To give better estimates, we may also want to state lower bounds on growth rates
- Definition: $T(n) = \Omega(f(n))$ if there exist constants c and n_0 such that $T(n) \geq cf(n)$ for all $n \geq n_0$
- Intuitively, an algorithm A is $\Omega(f(n))$ means that for (almost) all input sizes n , there is at least one instance of input of size n for which A requires (some constant fraction of) $f(n)$ time

Theta-Notation

- Definition: $T(n) = \Theta(f(n))$ if $T(n) = \Omega(f(n))$ and $T(n) = O(f(n))$
- Intuitively, an algorithm is $\Theta(f(n))$ means that $f(n)$ is a tight bound on its running time:
 - On all inputs of size n , time is $\leq f(n)$
 - On some input of size n , time is $\geq f(n)$

```
1 int k=0;
2 for ( int i=1; i<n; i*=2)
3     for ( int j=1; j<n; j++)
4         k++;
```

Above fragment is $O(n^2)$ but not $\Omega(n^2)$; it is $\Theta(n \log n)$

A more intricate analysis

```
1 int k=0;
2 for ( int i=1; i<n; i*=2)
3     for ( int j=1; j <= i; j++)
4         k++;
```

- Running time of inner loop: $O(i)$
- Assume $2^{l-1} < n \leq 2^l$, then total running time:

$$1 + 2 + 4 + \cdots + 2^{l-1} = 2^l - 1$$

- Running time is $\Theta(n)$

Exponentials vs. polynomials

- Goal: Establish cn^d grows slower than 2^n
- $cn^d = 2^{\log c} \cdot 2^{d \log n}$
- So $cn^d < 2^n$ if $\log c + d \log n < n$
- The function $n/(\log n)$ grows unbounded
- Choose n_0 such that

$$n_0/(\log n_0) > \log c + d$$

- For $n \geq n_0$, $cn^d < 2^n$
- E.g. for $n > 64$, $n/(\log n) > 10$ and hence,
 $n^{10} < 2^n$

Linear versus Binary Search

- Input: Array X contains n integers, already sorted in increasing order, and an integer x
- Output: Is x an element of the array?
- Linear search: Scan the array left to right
- Binary search: Locate the midpoint, decide whether x belongs to left half or right half, and repeat in the appropriate half
- Rule of thumb:
 - If problem size reduces by a constant by constant work, linear time
 - If size reduces by a constant fraction by constant work, logarithmic time

Linear Search

```
Linear_Search( Etype X[ ],
               Etype x, int n){
    for ( i=0; i<n; i++ ){
        if X[i] == x return i;
        if X[i] > x return Not_Found;
    }
    return Not_Found;
}
```

- Running time: $O(n)$
- Note: We do *worst-case* analysis
- Alternative: *average-case* analysis. In this case, it is $n/2$

Binary Search

```
Binary_Search( Etype X[ ],
               Etype x, int n){
    int low = 0, high = n-1;
    int mid;
    while ( low <= high ){
        mid = ( low + high ) /2;
        if ( X[mid] < x ) low = mid + 1;
        else if ( X[mid] > x ) high = mid - 1;
        else return mid;
    }
    return Not_Found;
}
```

- Running time of the loop : $O(1)$
- Initially $high - low$ is $n - 1$, and decreases by half in each iteration
- Total time: $O(\log n)$

Euclid's Algorithm

Classical algorithm to compute greatest common divisor efficiently

```
GCD ( int m, int n){
    int r;
    while ( n > 0 ){
        r = m % n; m = n; n = r;
    }
    return m;
}
```

Sample execution:

$$\begin{array}{ll} m = 1203, & n = 522 \\ m = 522, & n = 159 \\ m = 159, & n = 45 \\ m = 45, & n = 24 \\ m = 24, & n = 21 \\ m = 21, & n = 3 \\ m = 3, & n = 0 \end{array}$$

Analysis of Euclid's Algorithm

- Correctness: If $m > n > 0$ then

$$\text{GCD}(m, n) = \text{GCD}(n, m \bmod n)$$

- Thm: If $m > n$ then $m \bmod n < m/2$
- It follows that the remainder decreases by a factor of 2 every two iterations
- Number of iterations: $2 \log n$
- Running Time: $O(\log n)$

Computing Fibonacci numbers

Suppose you write the following program

```
long int Fib (n){  
    if ( n <= 1 ) return 1;  
    else return Fib ( n-1 ) + Fib ( n-2 );  
}
```

Try `Fib(100)`, and it takes forever.

What's wrong?

Try to analyze running time.

Exponential running time

- Running time T is defined by

$$T(0) = T(1) = c; T(n) = T(n-1) + T(n-2) + c$$

- Prove by induction: $T(n) \leq 2^n + 2^n c - c$
- It follows that running time is $O(2^n)$
- Similarly, we can show that running time is $\Omega((3/2)^n)$

Efficient Fibonacci Numbers

- Avoid recomputation
- Solution with linear running time

```
long int Fib (n){
    long int result [ Max_Inp ];
    if ( n >= Max_Inp ) return Error;
    result [0] = 1;
    result [1] = 1;
    for ( i=2; i<=n; i++)
        result [i] = result [i-1] + result [i-2];
    return result [n];
}
```

How does this relate to practice?

- We ignore many important effects that will determine the actual running time
 - Speed of the processor
 - Choice of the compiler
 - Constants are ignored
 - Fine-tuning by programmer
 - Different basic operations take different times
 - Peripheral effects: load, I/O, available memory
- In spite of above, $O(n)$ algorithm will outperform $O(n^2)$ algorithm for “large enough” input
- $O(2^n)$ algorithms will never work on large inputs

Maximum Subsequence Sum Problem

- Input: Array X of n integers (possibly negative)
- Output: Find a subsequence with maximum sum, that is, find $0 \leq i \leq j < n$ to maximize $\sum_{k=i}^j X[k]$
- Assumption: if all are negative, then output is 0
- Different algorithms with different running times

First Solution

- For every pair (i, j) with $0 \leq i \leq j < n$, compute the sum $\sum_{k=i}^j X[k]$

```
1  MSS1 (int X[ ], int n){
2    int current = 0, result = 0;
3    for ( int i=0; i<n; i++ )
4        for ( int j=i; j<n; j++ ){
5            current = 0;
6            for ( int k=i; k<=j; k++)
7                current += X[k];
8            if current > result
9                result = current;
10        }
11    return result;
12 }
```

Analysis of MSS1

- Three nested loops with constant work otherwise: $O(n^3)$
- Will more careful analysis give a better bound?
- Number of iterations of innermost loop (line 7) is $j - i + 1$
- Running time of lines 4–10:
$$\sum_{j=i}^{n-1} j - i + 1 = (n - i)(n - i + 1)/2$$
- Total running time:
$$\sum_{i=0}^{n-1} (n - i)(n - i + 1)/2 = (n^3 + 3n^2 + 2n)/6$$
- Running time is $\Theta(n^3)$

A Quadratic Solution

- Observation: Sum of $X[i..(j + 1)]$ can be computed by adding $X[j + 1]$ to sum of $X[i..j]$
- MSS2 has $\Theta(n^2)$ running time

```
1  MSS2 (int X[ ], int n){
2    int current = 0, result = 0;
3    for ( int i=0; i<n; i++ ){
4        current = 0;
5        for ( int j=i; j<n; j++ ){
6            current += X[j];
7            if current > result
8                result = current;
9        }
10   }
11   return result;
12 }
```

Recursive Solution

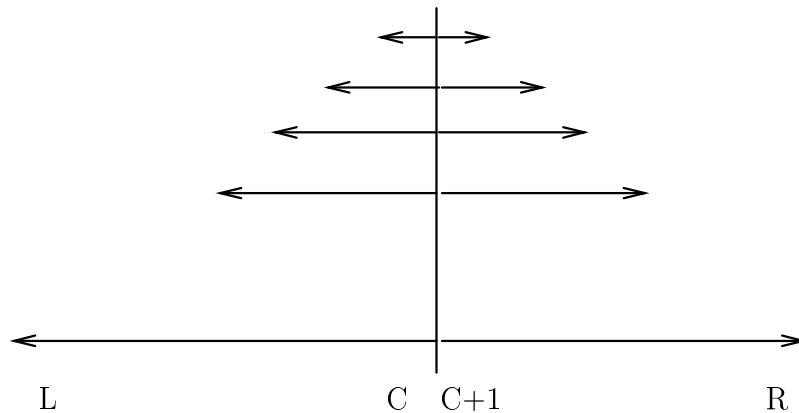
- Divide the problem in two parts: find maximum subsequences of left and right halves, and take the maximum of the two

```
MSS3 (int X[ ], int n){  
    return RMSS ( X, 0, n-1)}
```

```
RMSS (int X [], int L, int R ){  
    if ( L == R )    // base case  
        return ( max ( X[L] , 0 ));  
    int C = ( L + R ) /2;  
    int LS = RMSS ( X, L, C);  
    int RS = RMSS ( X, C+1, R);  
    return ( max ( LS, RS ));  
}
```

Is MSS3 correct?

- No! We did not consider the case when the desired subsequence spans both halves
- The missing case will include elements C and $C + 1$, and corresponds to



Max among these sums + Max among these sums

That is, compute:

$$\max_i \left\{ \sum_{k=i}^C X[k] \mid L \leq i \leq C \right\} +$$
$$\max_j \left\{ \sum_{k=C+1}^j X[k] \mid C + 1 \leq j \leq R \right\}$$

Correct RMSS

```
RMSS (int X [], int L, int R ){
    if ( L == R ) return ( max (X[L],0));
    int C = ( L + R ) /2;
    int LS = RMSS ( X, L, C);
    int RS = RMSS ( X, C+1, R);
    int current = result = X[C];
    for ( int i=C-1; i>=L; i--){
        current += X[i];
        result = max ( result, current );}
    current = result = result + X[C+1];
    for ( i=C+2; i<=R; i++){
        current += X[i];
        result = max ( result, current );}
    return ( max ( LS, RS, result ));
}
```

Analysis of MSS3

- Let $T(n)$ be the running time of RMSS for $L - R + 1 = n$
- Base case: $T(1) = O(1)$
- Recursive case:
 - Two recursive calls of size $n/2$
 - Plus $O(n)$ work for the added code
- This gives
$$T(1) = O(1); T(n) = 2T(n/2) + O(n)$$
- Check that for $n = 2^\ell$, $T(n) = n\ell + n$ satisfies the equation
- Running time $T(n) = n \log n + n = O(n \log n)$

Towards an Improved Solution

- Let us call position j a breakpoint if the sums $X[i..j]$ are negative for all $0 \leq i \leq j$

- Example:

2 6 - 3 - 7 5 - 2 4 - 12 9 - 4

- Property 1: Max subsequence won't include a breakpoint
- If j is a breakpoint, then solution is max of the solutions of the two halves $X[0..j]$ and $X[j + 1..n - 1]$
- Property 2: If j is the least position such that the sum $X[0..j]$ is negative, then j is a breakpoint

Final Solution

```
1 MSS4 (int X[ ], int n){
2   int current = 0, result = 0; \\ i=0
3   for ( int j=0; j<n; j++){
4       current += X[j];
5       result = max ( result, current );
6       if ( current < 0 )
7           current = 0 ;           \\ i=j+1
8   }
9   return (result);
}
```

- Single loop: running time is $O(n)$

Correctness of MSS4

1. $current = \sum_{k=i}^j X[k]$ after step 4
2. If $j > i$, then $\sum_{k=i}^{j-1} X[k] > 0$
3. If $current < 0$ then $X[j]$ is negative with $|X[j]| > \sum_{k=i}^{j-1} X[k]$
4. If $current < 0$, j is a breakpoint
 - Proof by induction on j using 3
5. $result$ equals maximum subsequence sum in $X[0..j]$ after step 5
 - Proof by induction on j using 4

Summary: Lower vs. Upper bounds

- Worst-case analysis that estimates running times
- Upper bound $O(f(n))$ means that for sufficiently large inputs, running time $T(n)$ is bounded by a multiple of $f(n)$
- Lower bound $\Omega(f(n))$ means that for sufficiently large n , there is at least one input of size n such that running time is at least a fraction of $f(n)$

Algorithms vs. Problems

- Running time analysis establishes bounds for individual algorithms
- Upper bound $O(f(n))$ for a *problem* means that there is *some* $O(f(n))$ *algorithm* to solve the problem
- Lower bound $\Omega(f(n))$ for a *problem* means that *every algorithm* to solve the problem is $\Omega(f(n))$
- While MSS2 is $\Omega(n^2)$, the max-subsequence-sum problem is not $\Omega(n^2)$ (because MSS4 solves it in $O(n)$)
- A mathematically difficult branch of algorithms: establishing lower bounds for problems (e.g. sorting requires $\Omega(n \log n)$ comparisons)

Summary: Recursion Analysis

1. Problem size is reduced by half using $O(1)$ work: $O(\log n)$ (e.g. binary search)
2. Problem is divided into two halves, each half is solved recursively, and results put together using $O(1)$ work: $O(n)$
3. Problem is divided into two halves, each half is solved recursively, and results put together using $O(n)$ work: $O(n \log n)$ (e.g. MSS3)
4. Problem is solved by two recursive calls, each smaller only by a constant amount: $O(2^n)$ (e.g. wasteful Fibonacci)