Practices for Time Complexity Analysis

Algorithms
Kyuseok Shim
SoEECS, SNU.

The Logarithm

- Formal Definition
  - For any B, $N > 0$, $\log_B N = K$ if $B^K = N$.
  - If (the base) B is omitted, it defaults to 2 in computer science.
- Examples:
  - log 32 = 5 (because $2^5 = 32$)
  - log 1024 = 10
  - log 1048576 = 20
  - log 1 billion = about 30
- The logarithm grows much more slowly than N, and slower than the square root of N.

Static Searching

- Given an integer *X* and an array *A*, return the position of *X* in *A* or an indication that it is not present. If *X* occurs more than once, return any occurrence. The array *A* is not altered.
- If input array is not sorted, solution is to use a sequential search. Running times:
  - Unsuccessful search: $O(N)$; every item is examined
  - Successful search:
    - Worst case: $O(N)$; every item is examined
    - Average case: $O(N)$; half the items are examined
- Can we do better if we know the array is sorted?

Binary Search

- Yes! Use a binary search.
- Look in the middle
  - Case 1: If *X* is less than the item in the middle, then look in the subarray to the left of the middle
  - Case 2: If *X* is greater than the item in the middle, then look in the subarray to the right of the middle
  - Case 3: If *X* is equal to the item in the middle, then we have a match
  - Base Case: If the subarray is empty, *X* is not found.
- This is logarithmic by the repeated halving principle.

Binary Search Continued

- Can do one comparison per iteration instead of two by changing the base case.
- See online code for details.
- Average case and worst case in revised algorithm are identical. 1 + log *N* comparisons (rounded down to the nearest integer) are used. Example: If *N* = 1,000,000, then 20 element comparisons are used. Sequential search would be 25,000 times more costly on average.
- Back to <u>interfaces</u>

Binary Search Algorithm

```
int binarySearch(int a[], int x)
{
    int low = 0, high = a.length - 1;
    while( low <= high )
    {
        int mid = ( low + high ) / 2;
        if( a[ mid ] < x )
             low = mid + 1;
        else if( a [ mid ] > x)
            high = mid – 1;
        else
            return mid;
    }
    return NOT_FOUND;
}
```

Binary Search

- Binary Search is an example of a data structure implementation:
  - *Insert*: $O(N)$ time per operation, because we must insert and maintain the array in sorted order.
  - *Delete*: $O(N)$ time per operation, because we must slide elements that are to the right of the deleted element over one spot to maintain contiguity.
  - *Find*: $O(\log N)$ time per operation, via binary search.
- In this course we examine different data structures. Generally we allow *Insert*, *Delete*, and *Find*, but *Find* and *Delete* are usually restricted. Example: in a stack, only last item is accessible.

Long pow(x, int n)

- Long pow(long x, int n)

Exponentiation – O(n)

```
 public static long pow( long x, int n )
{
    if( n == 0 )
        return 1;
     if( n == 1 )
        return x;
    return x*pow( x, n-1);
}
```

Exponentiation – O(lon n)

```
public static long pow( long x, int n )
{
    if( n == 0 )
        return 1;
     if( n == 1 )
        return x;
    if( isEven( n ) )
        return pow( x * x, n / 2 );
    else
        return pow( x * x, n / 2 ) * x;
}
```

Maximum Subsequence Sum Problem

- Examine a problem with several different solutions.
  - Will look at four algorithms
  - Some algorithms much easier to code than others
  - Some algorithms much easier to prove correct than others
  - Some algorithms much, much faster (or slower) than others

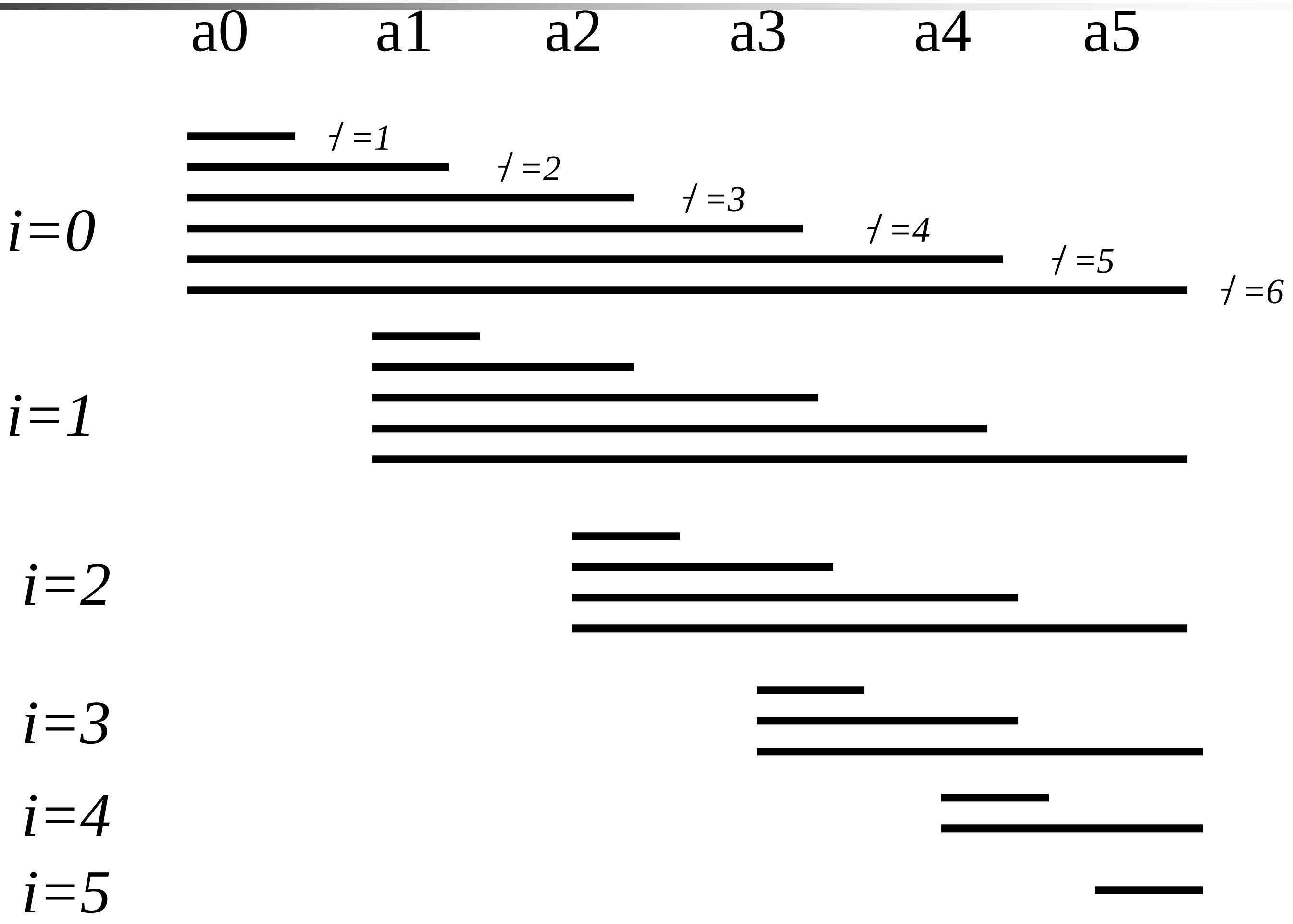
The Problem

- Maximum Contiguous Subsequence Sum Problem
  - Given (possibly negative integers) A_1, A_2, …, A_N, find (and identify the sequence corresponding to) the maximum value of $(A_i + A_{i+1} + \ldots + A_j)$.
- The maximum contiguous subsequence sum is zero if all the integers are negative. (Why?)
- Examples (maximum subsequences are underlined)
  - −2, $\underline{11, -4, 13}$, −4, 2
  - 1, −3, $\underline{4, -2, -1, 6}$

Brute Force Algorithm

```
int MaxSubSum1( const vector<int> & A)
{
    int MaxSum = 0;
    for( int i = 0; i < A.size(); i++ )
        for( int j = i; j < A.size(); j++ )
        {
            int ThisSum = 0;
            for( int k = i; k <= j; k++ )
                ThisSum += A[ k ];
            if( ThisSum > MaxSum )
                MaxSum = ThisSum;
        }
    return MaxSum;
}
```

Subsequence Generation in the Cubic Algorithm



Analysis

- Loop of size N inside of loop of size N inside of loop of size N means O(N^3), or cubic algorithm.
- Slight over-estimate (a factor of 6) that results from some loops being of size less than N is not important.

Actual Running Time

- For $N = 100$, actual time is 0.47 seconds on a particular computer.
- Can use this to estimate time for larger inputs:

$$T(N) = cN^3$$

$$T(10N) = c(10N)^3 = 1000cN^3 = 1000T(N)$$

- Inputs size increases by a factor of 10 means that running time increases by a factor of 1,000.
- For $N = 1000$, estimate an actual time of 470 seconds. (Actual was 449 seconds).
- For $N = 10{,}000$, estimate 449000 seconds (6 days).

How To Improve

- Remove a loop; not always possible.
- Here it is: innermost loop is unnecessary because it throws away information.
- `ThisSum` for next `j` is easily obtained from old value of `ThisSum`:
  - Need $A_i + A_{i+1} + \ldots + A_{j-1} + A_j$
  - Just computed $A_i + A_{i+1} + \ldots + A_{j-1}$
  - What we need is what we just computed + A_j

The Better Algorithm

```
int MaxSubSum2( const vector<int> &A)
{
    int MaxSum = 0;
    for( int i = 0; i < A.size(); i++ )
    {
        int ThisSum = 0;
        for( int j = i; j < A.size(); j++ )
        {
            ThisSum += A[ j ];
            if( ThisSum > MaxSum )
                MaxSum = ThisSum;
        }
    }
    return MaxSum;
}
```

Analysis

- Same logic as before: now the running time is quadratic, or $O(N^2)$
- As we will see, this algorithm is still usable for inputs in the tens of thousands.
- Recall that the cubic algorithm was not practical for this amount of input.

Actual running time

- For $N = 100$, actual time is 0.011 seconds on the same particular computer.
- Can use this to estimate time for larger inputs:

$$T(N) = cN^2$$

$$T(10N) = c(10N)^2 = 100cN^2 = 100T(N)$$

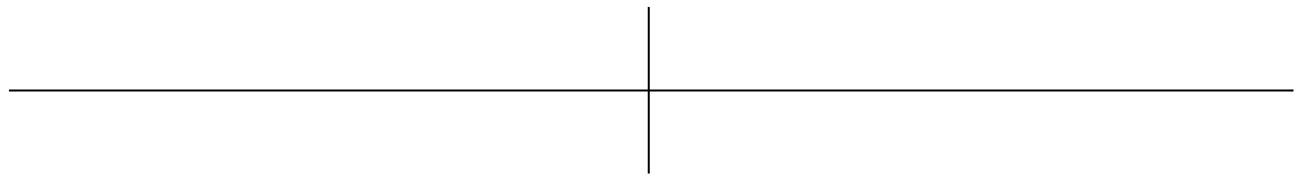
- Inputs size increases by a factor of 10 means that running time increases by a factor of 100.
- For $N = 1000$, estimate a running time of 1.11 seconds. (Actual was 1.12 seconds).
- For $N = 10{,}000$, estimate 111 seconds (= actual).

Recursive Algorithm

- Use a divide-and-conquer approach.
- The maximum subsequence either
  - lies entirely in the first half
  - lies entirely in the second half
  - starts somewhere in the first half, goes to the last element in the first half, continues at the first element in the second half, ends somewhere in the second half.
- Compute all three possibilities, and use the maximum.
- First two possibilities easily computed recursively.

Computing the Third Case

- Easily done with two loops; see the code
- For maximum sum that starts in the first half and extends to the last element in the first half, use a right-to-left scan starting at the last element in the first half.
- For the other maximum sum, do a left-to-right scan, starting at the first element in the first half.



Coding Details

- The code is more involved; see the online source.
- Make sure you have a base case that handles zero-element arrays.
- Use a public static driver with a private recursive routine.
- Recursion rules:
  - Have a base case
  - Make progress to the base case
  - Assume it works
  - Avoid computing the same solution twice

Analysis

- Let $T(N)$ = the time for an algorithm to solve a problem of size N.
- Then $T(1) = 1$ (1 will be the quantum time unit; remember that constants don't matter).
- $T(N) = 2\,T(N/2) + N$
  - Two recursive calls, each of size $N/2$. The time to solve each recursive call is $T(N/2)$ by the above definition
  - Case three takes $O(N)$ time; we use N, because we will throw out the constants eventually.

```
int maxSumRec( const vector<int> &A, int left, int right )
  {
       if( left == right )  // Base case
         if( A[left] > 0)
           return A[left];
         else
          return 0;

      int center = ( left + right ) / 2;
      int maxLeftSum  = maxSumRec( A, left, center );
      int maxRightSum = maxSumRec( A, center + 1, right );

      int maxLeftBorderSum = 0, maxRightBorderSum = 0;
      int leftBorderSum = 0, rightBorderSum = 0;
      for( int i = center; i >= left; i-- )   {
         leftBorderSum += A[ i ];
         if( leftBorderSum > maxLeftBorderSum )
             maxLeftBorderSum = leftBorderSum;
      }
      for( int i = center + 1; i <= right; i++ )    {
         rightBorderSum += A[ i ];
         if( rightBorderSum > maxRightBorderSum )
             maxRightBorderSum = rightBorderSum;
      }


      return max3( maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum );
  }
```

Bottom Line

$T(1) = 1 = 1 * 1$

$T(2) = 2 * T(1) + 2 = 4 = 2 * 2$

$T(4) = 2 * T(2) + 4 = 12 = 4 * 3$

$T(8) = 2 * T(3) + 8 = 32 = 8 * 4$

$T(16) = 2 * T(4) + 16 = 80 = 16 * 5$

$T(32) = 2 * T(5) + 32 = 192 = 32 * 6$

$T(64) = 2 * T(6) + 64 = 448 = 64 * 7$

$T(N) = N(1 + \log N) = O(N \log N)$

N log *N*

- Any recursive algorithm that solves *two half-sized problems* and does *linear non-recursive work* to combine/split these solutions will always take $O(N \log N)$ time because the above analysis will always hold.
- This is a very significant improvement over quadratic.
- It is still not as good as $O(N)$, but is not that far away either. There is a linear-time algorithm for this problem; see the online code. The running time is clear, but the correctness is non-trivial.
- Space Complexity?

The Logarithm

- Formal Definition
  - For any B, $N > 0$, $\log_B N = K$ if $B^K = N$.
  - If (the base) B is omitted, it defaults to 2 in computer science.
- Examples:
  - log 32 = 5 (because $2^5 = 32$)
  - log 1024 = 10
  - log 1048576 = 20
  - log 1 billion = about 30
- The logarithm grows much more slowly than N, and slower than the square root of N.

Static Searching

- Given an integer *X* and an array *A*, return the position of *X* in *A* or an indication that it is not present. If *X* occurs more than once, return any occurrence. The array *A* is not altered.
- If input array is not sorted, solution is to use a sequential search. Running times:
  - Unsuccessful search: $O(N)$; every item is examined
  - Successful search:
    - Worst case: $O(N)$; every item is examined
    - Average case: $O(N)$; half the items are examined
- Can we do better if we know the array is sorted?

Binary Search

- Yes! Use a binary search.
- Look in the middle
  - Case 1: If *X* is less than the item in the middle, then look in the subarray to the left of the middle
  - Case 2: If *X* is greater than the item in the middle, then look in the subarray to the right of the middle
  - Case 3: If *X* is equal to the item in the middle, then we have a match
  - Base Case: If the subarray is empty, *X* is not found.
- This is logarithmic by the repeated halving principle.

Binary Search Continued

- Can do one comparison per iteration instead of two by changing the base case.
- See online code for details.
- Average case and worst case in revised algorithm are identical. 1 + log *N* comparisons (rounded down to the nearest integer) are used. Example: If *N* = 1,000,000, then 20 element comparisons are used. Sequential search would be 25,000 times more costly on average.
- Back to interfaces

Binary Search Algorithm

```
int binarySearch(int a[], int x)
{
    int low = 0, high = a.length - 1;
    while( low <= high )
    {
        int mid = ( low + high ) / 2;
        if( a[ mid ] < x )
             low = mid + 1;
        else if( a [ mid ] > x)
            high = mid – 1;
        else
            return mid;
    }
    return NOT_FOUND;
}
```

Binary Search

- Binary Search is an example of a data structure implementation:
  - *Insert*: *O* (*N*) time per operation, because we must insert and maintain the array in sorted order.
  - *Delete*: *O* (*N*) time per operation, because we must slide elements that are to the right of the deleted element over one spot to maintain contiguity.
  - *Find*: *O*(log *N*) time per operation, via binary search.
- In this course we examine different data structures. Generally we allow *Insert*, *Delete*, and *Find*, but *Find* and *Delete* are usually restricted. Example: in a stack, only last item is accessible.

Long pow(x, int n)

- Long pow(long x, int n)

Exponentiation – O(n)

```
public static long pow( long x, int n )
{
     if( n == 0 )
         return 1;
     if( n == 1 )
         return x;
     return pow( x * x, n / 2 );
}
```

Exponentiation – O(lon n)

```
 public static long pow( long x, int n )
{
     if( n == 0 )
         return 1;
      if( n == 1 )
         return x;
     if( isEven( n ) )
         return pow( x * x, n / 2 );
     else
         return pow( x * x, n / 2 ) * x;
 }
```