CS130A

Discussion 1

06/22/2011

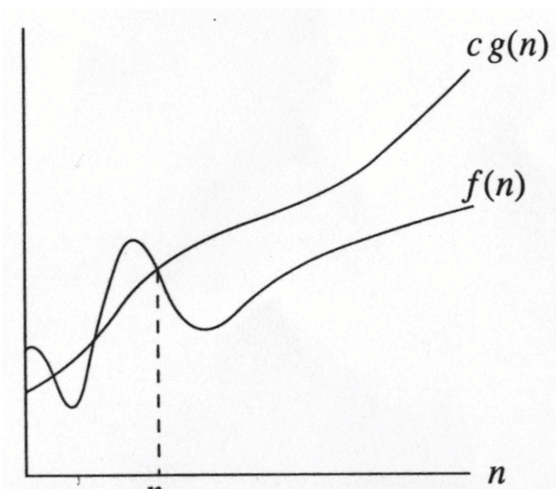
Outline

- Time complexity
  - Notation
  - Maximum subsequence sum problem
    - 4 algorithms ($O(N^3)$, $O(N^2)$, $O(N\log N)$, $O(N)$)
- Tries

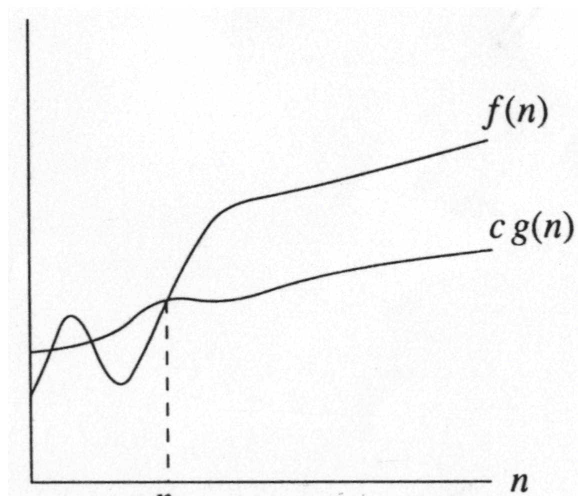
- In class exercise

Notation

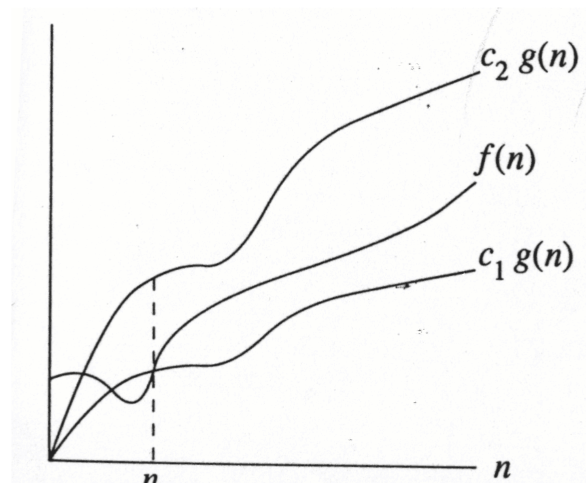
Asymptotically less than or equal to	O	(Big-Oh)
Asymptotically greater than or equal to	Ω	(Big-Omega)
Asymptotically equal to	θ	(Big-Theta)



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$

Review on typical growth rates

- Examples:
  - N^2
    $\log N$
    $(\log N)^2$
    $N \log N$
    N
    N^3
    2^N
    C

Function	Name
C	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Max Subsequence Problem

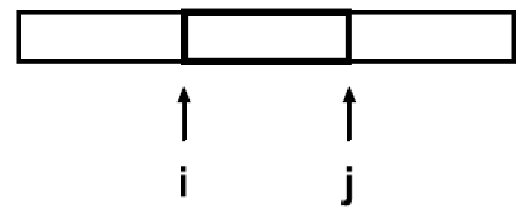
- Given a sequence of integers A1, A2, …, An, find the maximum possible value of a subsequence Ai, …, Aj.
- Numbers can be negative.
- You want a contiguous chunk with largest sum.

- Example: -2, 11, -4, 13, -5, -2
- The answer is 20 (subseq. A2 through A4).

- We will discuss 4 different algorithms, with time complexities $O(n^3)$, $O(n^2)$, $O(n \log n)$, and $O(n)$.
- With $n = 10^6$, algorithm 1 may take > 10 years; algorithm 4 will take a fraction of a second!

Algorithm 1 for Max Subsequence Sum

- Given $A_1,\ldots,A_n$, find the maximum value of $A_i+A_{i+1}+\cdots+A_j$
  0 if the max value is negative




```
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;
```

$O(1)$

$O(1)$

$O(1)$

$O(1)$

1 2 3

- Time complexity: $O(n^3)$

$$\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}\sum_{k=i}^{j} 1$$

1 $$\sum_{k=i}^{j} 1 = j-i+1$$

2 $$\sum_{j=i}^{N-1}(j-i+1) = \frac{(N-i+1)(N-i)}{2}$$

3 $$\sum_{i=0}^{N-1}\frac{(N-i+1)(N-i)}{2} = \frac{N^3+3N^2+2N}{6}$$

Algorithm 2

- Idea: Given sum from i to j-1, we can compute the sum from i to j in constant time.

$$\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$$

- This eliminates one nested loop, and reduces the running time to $O(n^2)$.

```
into 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;
```

Algorithm 3

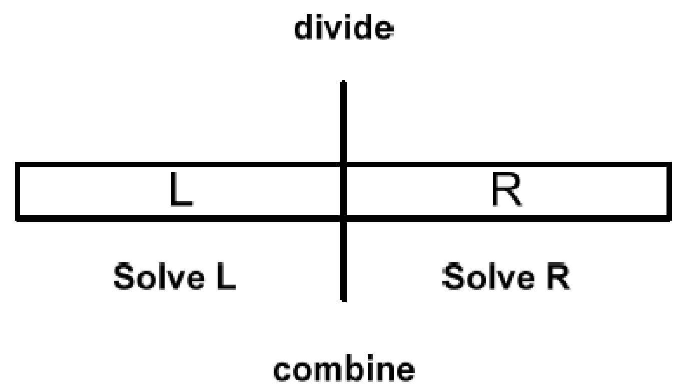
- This algorithm uses divide-and-conquer paradigm.
- Suppose we split the input sequence at midpoint.
- The max subsequence is entirely in the left half, entirely in the right half, or it cross the midpoint
  - If it spans the middle, then it includes the max subsequence in the left ending at the center and the max subsequence in the right starting from the center

Algorithm 3 (cont.)

- Maximum subsequence can be
  - In Left
  - In Right

  Solved recursively

  - In the middle:
    - Largest sum in L ending with middle element + largest sum in R beginning with middle element



- Example:

  left half | right half

  4 -3 5 -2 | -1 2 6 -2

- Max in left is 6 (A1 through A3); max in right is 8 (A6 through A7). But crossing max is 11 (A1 thru A7)

Algorithm 3 (cont.)

```
static int
MaxSubSum( const int A[ ], int Left, int Right )
{
    int MaxLeftSum, MaxRightSum;
    int MaxLeftBorderSum, MaxRightBorderSum;
    int LeftBorderSum, RightBorderSum;
    int Center, i;

    if( Left == Right )  /* Base Case */
        if( A[ Left ] > 0 )
            return A[ Left ];
        else
            return 0;

    Center = ( Left + Right ) / 2;
    MaxLeftSum = MaxSubSum( A, Left, Center );
    MaxRightSum = MaxSubSum( A, Center + 1, Right );

    MaxLeftBorderSum = 0; LeftBorderSum = 0
    for( i = Center; i >= Left; i-- )
    {
        LeftBorderSum += A[ i ];
        if( LeftBorderSum > MaxLeftBorderSum )
            MaxLeftBorderSum = LeftBorderSum;
    }

    MaxRightBorderSum = 0; RightBorderSum = 0;
    for( i = Center + 1; i <= Right; i++ )
    {
        RightBorderSum += A[ i ];
        if( RightBorderSum > MaxRightBorderSum )
            MaxRightBorderSum = RightBorderSum;
    }

    return Max3( MaxLeftSum, MaxRightSum,
                 MaxLeftBorderSum + MaxRightBorderSum );
}
```

Algorithm 3: Analysis

- Let $T(n)$ be the time it takes to solve for a maximum subsequence sum problem of size n
- The divide and conquer is best analyzed through recurrence:

$$T(1) = 1 \qquad \text{//constant time}$$
$$T(n) = 2T(n/2) + O(n)$$

- This recurrence solves to $T(n) = O(n \log n)$.

Algorithm 4

```
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); j++ )
    {
        thisSum += a[ j ];

        if ( thisSum > maxSum )
            maxSum = thisSum;
        else if ( thisSum < 0 )
            thisSum = 0;
    }
    return maxSum;
}
```

The algorithm resets whenever prefix is < 0. Otherwise, it forms new sums and updates maxSum in one pass.

- Time complexity clearly $O(n)$
- But why does it work?

Intuition

- One observation is that if a[i] is negative, then it cannot possibly be the start of the optimal subsequence, since any subsequence that begins with a[i] would improved by beginning with a[i+1]
  - Ex: -2 11 -4 13 -5 -2

- Similarly any negative subsequence cannot possibly be a prefix of the optimal subsequence (same logic)

- If we detect that the subsequence from a[i] to a[j] is negative in the inner loop, we can advance i. The crucial thing is that not only we can advance i to i+1, but all the way to j+1

Outline

- Time complexity
  - Notation
  - Maximum subsequence sum problem
    - 4 algorithms ($O(N^3)$, $O(N^2)$, $O(N\log N)$, $O(N)$)
- Tries
- In class exercise

Trie

- Prefix tree
  - an ordered tree data structure that is used to store an associative array where the keys are usually strings

- Time to insert, or to delete or to find is almost identical because the code paths followed for each are almost identical

- More space efficient when they contain a large number of short keys, because nodes are shared between keys with common initial subsequences.

- Slower if the data is directly accessed on a hard disk drive or some other secondary storage device

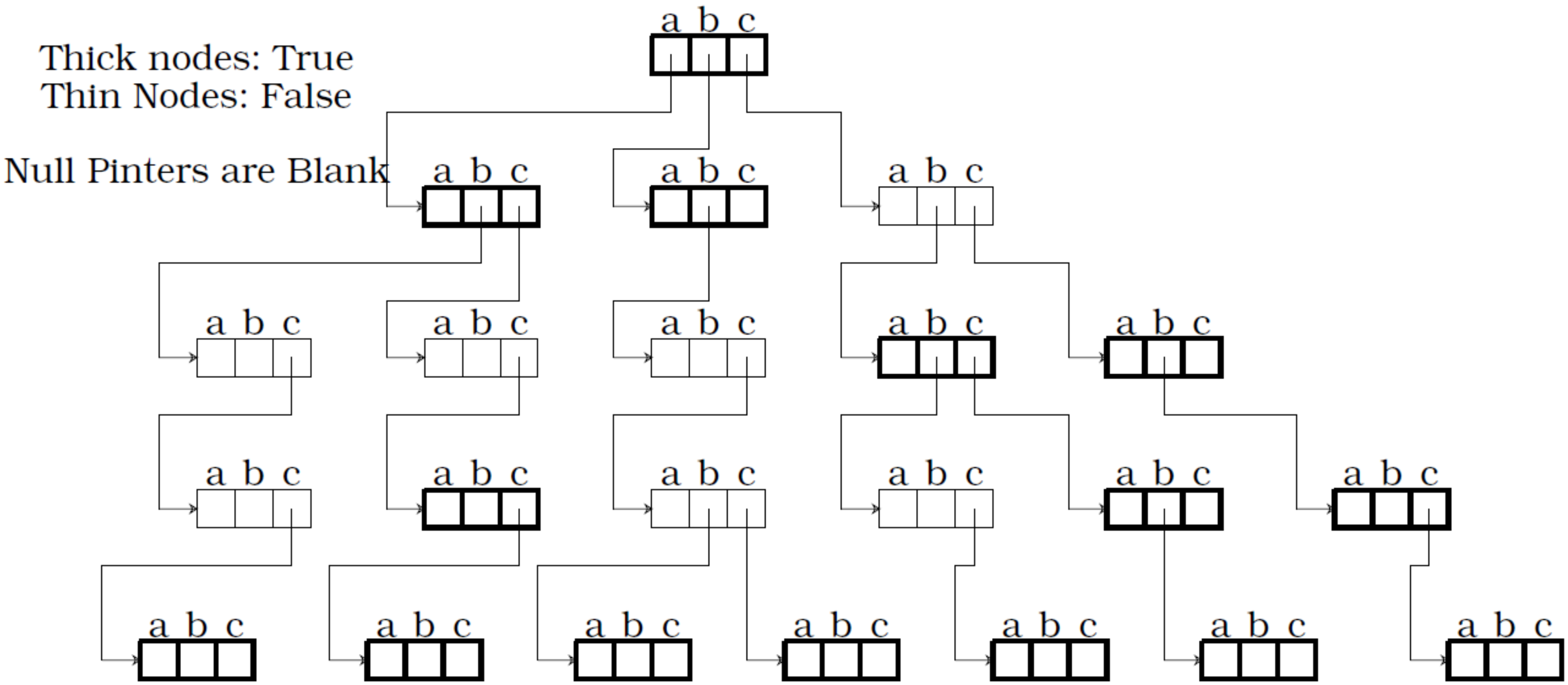
Trie implementation

```
class TrieNode {
private:
        bool StrEnds;
        TrieNode *ptr[TrieMaxElem];
public:
        TrieNode();
        void SetStrEnds(){StrEnds = true;}
        void UnSetStrEnds(){StrEnds =
false;}
        bool GetStrEnds(){return StrEnds;}
        void SetPtr(int i, TrieNode*
j){ptr[i]=j;}
        TrieNode* GetPtr(int i){return ptr[i];}
};
```

```
class Trie {
public:
        Trie() ;
        void Readlist();
        void Insert(char x[]);
        bool Member(char x[]);
        void Delete(char x[]);
private:
        TrieNode *root;
        bool Delete(char x[], int i,
        TrieNode *current );
        bool CheckTrieNodeEmpty(TrieNode
*current);
};
```

Trie example

$$\{a, abcc, acc, accc, b, bbcb, bbcc, cb, cbbc, cbc, cbcb, cc, ccb, ccbc\}$$



Trie Delete

```
bool Trie::Delete(char x[], int i, TrieNode *current){
  if (current != 0)
  {if (x[i] == '\0')                                          //if at the end of a string
         {current -> UnSetStrEnds();
          if (CheckTrieNodeEmpty(current))
                   {delete current; return true;} }
     else {
              if (Delete(x,i+1,current->GetPtr(x[i] - 'a')))
              {
                 current->SetPtr(x[i] - 'a', 0);     //set the entry of current node to Null
                 if (i != 0 && CheckTrieNodeEmpty(current))  //not to delete the root
                 {delete current; return true;}
              }
         }
  }
  return false;
}
```

Outline

- Time complexity
  - Notation
  - Maximum subsequence sum problem
    - 4 algorithms ($O(N^3)$, $O(N^2)$, $O(N \log N)$, $O(N)$)

- Tries

- In class exercise

Reference

- Data structure and algorithm analysis in C++ (3rd)
- Professor Suri's lecture note
  - http://www.cs.ucsb.edu/~suri/cs130a/cs130a.html
- Professor Qu's lecture note
  - http://www.cs.ust.hk/~huamin/COMP171/index.htm
- Lara Deek's slide on TrieNode and Trie