CS 341: ALGORITHMS

Lecture 3: reducing one problem to another

Slides by Trevor Brown (some material from Doug Stinson’s slides)

trevor.brown@uwaterloo.ca
(http://tbrown.pro)
Olive Garden waiter: Sir, you've already had 5 baskets of breadsticks
Me:

We're done when I say we're done
EXAMPLE 3  (BENTLEY’S PROBLEM, SOLUTION 1)

max := 0;
for i := 1 to n do
    for j := i to n do
        sum := 0;
        for k := i to j do
            sum := sum + A[k];
        if sum > max then max := sum;
Strategy 1: big-O and big-Ω bounds

\[ T(n) \in \Theta(1) + \sum_{i=1}^{n} \sum_{j=i}^{n} \left( \Theta(1) + \sum_{k=i}^{j} \Theta(1) + \Theta(1) \right) \]

\[ T(n) \in \sum_{i=1}^{n} \sum_{j=i}^{n} \Theta(j - i) \in \Theta \left( \sum_{i=1}^{n} \sum_{j=i}^{n} j - i \right) \]

\[ T(n) \in O \left( \sum_{i=1}^{n} \sum_{j=i}^{n} j - i \right) \leq O \left( \sum_{i=1}^{n} \sum_{j=i}^{n} n \right) \]

\[ \leq O \left( \sum_{i=1}^{n} \sum_{j=1}^{n} n \right) \]

\[ T(n) \in O(n^3) \]

This is the maximum number of iterations that could be performed in this loop.
Proving a big-Ω bound...

Recall:

\[ T(n) \in \Theta \left( \sum_{i=1}^{n} \sum_{j=i}^{n} (j - i) \right) \]

\[ T(n) \in \Omega \left( \sum_{i=1}^{n/2} \sum_{j=i}^{n} (j - i) \right) \]

\[ \geq \Omega \left( \sum_{i=1}^{n/2} \sum_{j=3n/4}^{n} (j - i) \right) \]

Intuition: \( j - i \) is \( \Omega(n) \) in some iterations. How many iterations? Lots?

To get a good \( \Omega \)-bound, we ask questions like:

- When do our loops have many iterations?
- When is our dominant term large?

Many iterations: when our \( j \) loop does \( \Omega(n) \) iterations! For example, when \( i \leq n/2 \)

Large dominant term: when \( j \) is much larger than \( i \) (i.e., by a factor of \( n \))

Max := 0;
for \( i := 1 \) to \( n \) do
  for \( j := i \) to \( n \) do
    sum := 0;
    for \( k := i \) to \( j \) do
      sum := sum + A[k];
    if sum > max then max := sum;
Proving a big-$\Omega$ bound... continued

Recall:

\[ T(n) \in \Omega \left( \sum_{i=1}^{n/2} \sum_{j=3n/4}^{n} j - i \right) \]

\[ \geq \Omega \left( \sum_{i=1}^{n/2} \sum_{j=3n/4}^{n} 3n/4 - n/2 \right) \]

\[ = \Omega \left( \sum_{i=1}^{n/2} \sum_{j=3n/4}^{n} n/4 \right) \]

\[ \geq \Omega \left( \frac{n}{2} \cdot \frac{n}{4} \cdot \frac{n}{4} \right) = \Omega(n^3) \]

Smallest possible value of $j - i$ for these bounds on $i, j$

We will perform at least this much work in every iteration!

This term does not depend on the loop indexes, so just multiply by the total number of loop iterations...

Since we have $O(n^3)$ and $\Omega(n^3)$, we have proved $\Theta(n^3)$

```
max := 0;
for i := 1 to n do
  for j := i to n do
    sum := 0;
    for k := i to j do
      sum := sum + A[k];
      if sum > max then max := sum;
```
A SIMPLE EXAMPLE PROBLEM (FOR LEARNING REDUCTIONS)
3SUM PROBLEM

• Input: Array $A = [A[1], ..., A[n]]$ of distinct integers
• Output: true if there exist three distinct values in $A$ whose sum equals 0, else false

Additional definitions:
A yes-instance is an input to a decision problem, for which the correct output is true
A no-instance is an input to a decision problem, for which the correct output is false

Since the output is true/false, this is called a “decision problem”
**SIMPLE (BRUTE FORCE) SOLUTION**

```c
1 3SUM BruteForce(int A[1..n])
2     for i = 1 .. n-2
3         for j = i+1 .. n-1
4             for k = j+1 .. n
6                     return true
7     return false
```

Runtime $\Theta(n^3)$ by similar arguments to earlier...

Idea: let’s turn the innermost loop into something more efficient...
AN IMPROVEMENT


• Instead of looping over $k$, search the array for $-(A[i] + A[j])$

How to do this efficiently?
IMPROVED ALGORITHM

• Use binary search:
  • Searches $n$ elements in $O(\log n)$ time
  • Requires elements to be sorted!

```c
3SUM_Improved(int A[1..n])
    sort(A)
    for i = 1 .. n-2
        for j = i+1 .. n-1
            k = binary search for -(A[i]+A[j]) in the subarray A[j+1..n]
            if search is successful return true
    return false
```

VS. linear search, which takes $O(n)$ time

What is this algorithm’s time complexity?
TIME COMPLEXITY

\[ O(n \log n) \]

\[ O(\log(\text{subarray size})) = O(\log(n - (j + 1) + 1)) \leq O(\log n) \]

- **Inner loop:** iterations * work per iteration
  - \( O(n) \times O(\log n) = O(n \log n) \)
- **Outer loop:** \( O(n) \) * inner loop
  - \( O(n) \times O(n \log n) = O(n^2 \log n) \)
- **Entire algorithm:** \( O(n \log n) + O(n^2 \log n) = O(n^2 \log n) \)

\[ O(n) \text{ iterations} \]

\[ O(n) \text{ iterations} \]
PREPROCESSING

• The sort is an example of pre-processing
• It modifies the input to permit a more efficient algorithm (binary search as opposed to linear search)
• Note that a pre-processing step is only done once
FURTHER IMPROVEMENT

• Can actually improve to $O(n^2)$ time with a \textit{greedy} approach, but we won’t cover that for now...
REDUCTIONS: TRANSFORMING ONE PROBLEM INTO ANOTHER

“That’s Briggs, our new department head. He’s got an amazing knack for reducing complex problems into easy-to-understand witch hunts!”
REDUCTIONS

• Suppose $P$ and $Q$ are problems, and $\text{SolveQ}$ is a hypothetical algorithm that solves $Q$

• Suppose you design an algorithm $\text{ReducePtoQ}$ that solves $P$, and this algorithm calls $\text{SolveQ}$ as a subroutine

1. \text{ReducePtoQ}(P\_args)
2. \hspace{1em} $Q\_args = \text{PreprocessInput}(P\_args)$
3. \hspace{1em} return $\text{SolveQ}(Q\_args)$

• $\text{ReducePtoQ}$ is called a reduction from $P$ to $Q$

• It reduces $P$ to the problem of solving $Q$

• If $P$ can be reduced to $Q$, we denote this by $P \leq Q$

Mnemonic: $Q$ goes into $P$ as a subproblem.
A SIMPLE REDUCTION

• Suppose we want to multiply two integers, \( x \) and \( y \)

• Consider the algebraic identity: \( xy = \frac{(x+y)^2-(x-y)^2}{4} \)

• This allows us to show that \text{Multiplication} \leq \text{Squaring}

\begin{verbatim}
1   ReduceMultiplyToSquare(int x, int y)
2     s = ComputeSquare(x+y)
3     t = ComputeSquare(x-y)
4     return ((s-t)/4)
\end{verbatim}

• **Oracle**: ComputeSquare
  
  • Oracle “gives” you a solution to the **subproblem**...
  
  • If **you** solve ComputeSquare, you’ve solved **Multiply**

Note that “division by 4” here is just bit-shifting by 2.
A MORE ADVANCED REDUCTION

• **Target3SUM** problem
  • Input: an array $A$ of $n$ distinct integers, and a **target integer** $T$
  • Output: true if there are three distinct elements in $A$ whose sum equals $T$, else false

• It is straightforward to **modify** any algorithm for **3SUM** so it solves **Target3SUM**

• Another approach is to find a **reduction** **Target3SUM $\leq$ 3SUM**. This would allow code re-use.
TARGET3SUM ≤ 3SUM

- if and only if $3A[i] + 3A[j] + 3A[k] - 3T = 0$

- This suggests the following approach

```plaintext
1. Reduce_Target3SUM_to_3SUM(A, T)
   for i = 1..n
     B[i] = 3*A[i] - T
   return Solve3SUM(B)
```

Preprocess input to create new array B

Use the oracle to solve subproblem 3SUM(B)
If this reduction is correct, the result should also be a solution to problem Target3SUM(A, T)
COMPLEXITY ANALYSIS

Suppose we implement the oracle \texttt{Solve3SUM} with one of our “real” \texttt{3SUM} algorithms.

What is the complexity of the resulting “complete” algorithm?

If we plug in \texttt{3SUM\_BruteForce}, we get $O(n) + O(n^3) = O(n^3)$.

If we plug in \texttt{3SUM\_Improved}, $O(n) + O(n^2 \log n) = O(n^2 \log n)$.

```python
1 Reduce_Target3SUM_to_3SUM(A, T)
2   for i = 1..n
3     B[i] = 3*A[i] - T
4   return Solve3SUM(B)
```

$O(n)$ preprocessing work

+ cost of \texttt{Solve3SUM}
MANY-ONE REDUCTIONS

• The previous reduction had a very special structure
  • We transformed (reduced) an instance of the first problem to an instance of the second problem
  • We called the oracle once, on the transformed instance
• Reductions of this form, in the context of decision problems, are called many-one reductions
  • (also known as polynomial transformations or Karp reductions)
• We will many examples of these in the section on intractability
A MORE ADVANCED REDUCTION

• **3array3SUM** problem
  - Input: three arrays of n distinct integers: \( A, B \) and \( C \)
  - Output: true if there exist \( A[i], B[j], C[k] \), whose sum equals 0, else false

• Let’s try to reduce this to 3SUM

```python
1 TRY_Reduce_3array3SUM_to_3SUM(A, B, C)
2 A' = concatenation of A, B and C
3 return 3SUM(A')
```

Is this reduction correct?

Problem: **3SUM** might choose \( \geq 2 \) elements from the same array!

How can we ensure that **3SUM** picks one element from each subarray?
A BETTER ATTEMPT

• Input: three arrays of n distinct integers: A, B and C
• Output: true if there exist $A[i], B[j], C[k]$, whose sum equals 0, else false

```python
Reduce_3array3SUM_to_3SUM(A, B, C):
    for i = 1 .. n-1
        D[i] = 10A[i] + 1
        E[i] = 10B[i] + 2
        F[i] = 10C[i] - 3
    A' = concatenation of D, E and F
    return 3SUM(A')
```

How to argue 3SUM picks one element per sub-array?
CORRECTNESS OF THE REDUCTION (1/3)

• To show that this reduction is correct, we prove:
• **true** is the correct output for \(3SUM(A')\) if and only if
• **true** is the correct output for \(\text{Reduce}_3\text{array}_3\text{SUM}_3\text{to}_3\text{SUM}(A, B, C)\)

```plaintext
1  Reduce_3array_3SUM_to_3SUM(A, B, C)
2     for i = 1..n-1
3         D[i] = 10A[i] + 1
4         E[i] = 10B[i] + 2
5         F[i] = 10C[i] - 3
6     A' = concatenation of D, E and F
7     return 3SUM(A')
```
CORRECTNESS OF THE REDUCTION (2/3)

• To show that this reduction is correct, we prove:
  true is the correct output for $3\text{SUM}(A')$ if and only if
  true is the correct output for $3\text{array3SUM}(A, B, C)$
  
• Case 1: Assume true is the correct output for $3\text{array3SUM}(A, B, C)$
  
  • Want to show true is the correct output for $3\text{SUM}(A')$
  
  • By our assumption, there exist $A[i] + B[j] + C[k] = 0$
  
  • So $10A[i] + 1 + 10B[j] + 2 + 10C[k] - 3 = 0$

  \[ D[i] + E[j] + F[k] \]

  • So true is the correct output for $3\text{SUM}(A')$

Recall $D$, $E$ and $F$ are concatenated to get $A'$
Then the sum $A'[i] + A'[j] + A'[k] + 10B[...] + 2 + 10B[...] + 2 + 10B[...] + 2$, which is not zero! Contradiction!

So, there is a sum=0, with one from each of $A, B$ and $C$. So, true is the correct output for $3array3SUM(A, B, C)$.