

Divide and conquer and dynamic programming



Presentation by Antonio-Gabriel Sturzu

- Divide and conquer
 - General presentation
 - Raising to a power in logarithmic time
 - Maximum subarray problem
 - Karatsuba multiplication
- Dynamic programming
 - General presentation
 - Problems

- Has three basic steps:
 - Divide step
 - Conquer step
 - Combine step
- In the divide step you divide the problem into smaller subproblems
- In the conquer step you recursively solve those problems

- In the combine step you use the solutions for smaller subproblems in order to solve the initial problem
- When the subproblems are small enough you can solve them in a straightforward manner
- In general the problems do not overlap
- Examples:
 - Binary search
 - MergeSort

- Problems
- Raising to a power in logarithmic time
- Let's say we want to compute the last digit of a^b
- This is equivalent to $a^b \bmod 10$
- How to apply divide and conquer here ?

- There are two cases
- b even or odd
- If b is even we can write $a^b = a^{b/2} * a^{b/2}$
- If b is odd we can write $a^b = a * a^{b/2} * a^{b/2}$
- This is all we have to do

- Pseudocode:

- `int pow(int a, int b)`

- {

- `if(b==1)`

- `return a%10;`

- `if(b%2==0)`

- {

- `return (pow(a,b/2)*pow(a,b/2))%10;`

- }

– `return ((a%10)*((pow(a,b/2)*pow(a,b/2))%10))%10;`

 `}`

 This code is correct but has a problem

 So what is the problem with this code ?

- The problem is the time complexity
- If we assume that the multiplication is done in $O(1)$ then the presented pseudocode has the following time complexity recurrence:
 - $T(n) = 2 * T(n/2) + O(1)$
 - This gives $\theta(n)$ time complexity which is not better than the trivial algorithm

- Because the problems overlap the solution is to not recompute the results for things that we already know
- In this case this only requires an additional variable per recursion level
- At each recursion level we keep the result for $\text{pow}(a, b/2)$ and in this way the time complexity recurrence relation becomes $T(n) = T(n/2) + \theta(1)$ which gives the time complexity of $\theta(\log_2 n)$

- Given an array of integer numbers find the subsequence with the maximum sum
- A subsequence is a contiguous sequence of the original array
- Example:
 - -1, -4, -5 , 10, 20, -4, 30
 - The solution here is 10, 20, -4, 30

- How to apply the divide and conquer paradigm here ?
- The intuitive idea would be to just break up the array in two and recursively solve the two smaller subarrays
- The main problem here is the combine step
- How to perform the combine step ?

- We know the solution for the left subarray and for the right subarray
- But there might exist a solution that uses elements from both the left subarray and right subarray
- How to find it and what time complexity should it have ?



Solution:

- The key idea is that we have an additional constraint that helps us
- The middle points must be included in the crossing subarray
- Let low, mid and high be the left end, middle and right end of the array



Solution:

- We can find the maximum sum of the crossing subarray in the following manner:
 - Find the maximum sum in the left subarray that starts from mid and goes down to low
 - Find the maximum sum in the right subarray that starts from mid+1 and goes up to right
 - The subarray must be contiguous



Solution:

- We can do this in a straightforward manner in linear time !
- Pseudocode:
 - `int findcrossmax(A, int low, int mid, int high)`
 - {
 - » `int sum=0;`
 - » `int left=A[mid];`

- `for(i=mid;i>=low; --i)`
- `{`
 - `sum+=A[i];`
 - `if(sum>left)`
 - » `left=sum`
- `}`
- `sum=0;`
- `int right=A[mid+1];`

- `for(i=mid+1;i<=high;++i)`
- `{`
 - `sum+=A[i];`
 - `if(sum>right)`
 - » `right=sum;`
- `}`
- `return left+right;`

- The solution for the main problem is simple

- Pseudocode:

- `int findmax(A, int left, int right)`
- `{`
 - `if(left==right)`
 - » `return A[left];`
 - `int mid=left+(right-left)/2;`
 - `int st=findmax(A,left,mid);`
 - `int dr=findmax(A,mid+1,right);`

Pseudocode:

- {
 - `int cross=findcrossmax(A,left,mid,right);`
 - `return max(st,dr,cross);`
- }

Time complexity ?

- The recurrence relation is $T(n) = 2 \cdot T(n/2) + \theta(n)$
- The same as mergesort so the time complexity is $\theta(n \log n)$ better than the $\theta(n^2)$ optimized brute force algorithm

- Karatsuba multiplication of BIG integer numbers
- If the two numbers to be multiplied have length n each then the algorithm learned in school will take $\theta(n^2)$ time
- How can we use the divide and conquer paradigm here ?

- Solution
- Let the the two numbers be n_1 and n_2
- The basic idea is to divide the two numbers in two approximately equal length numbers
- This can be done by writing the numbers in a specific base, for example base 10

- $n_1 = 10^{n/2}b + c, n_2 = 10^{n/2}d + e$
- $n_1 n_2 = 10^n bd + 10^{n/2}be + 10^{n/2}cd + ce$
- In this formula four multiplications of $n/2$ length numbers are done and also four additions of numbers whose length is at most n
- The time complexity recurrence will be:
- $T(n) = 4T(n/2) + \theta(n)$

- By using the master theorem this yields a time complexity of $\theta(n^2)$ which is the same as the school algorithm
- In order to reduce the time complexity we must reduce the number of multiplications which are done
- Let's write $n_1 n_2 = 10^n bd + 10^{n/2}(be + cd) + ce$

- The key idea is that in order to compute $be+cd$ we don't need to compute be and cd
- We will only compute $(b+c)(d+e)$
- Here's the complete formula:
- $n_1 n_2 = 10^n bd + 10^{n/2}[(b+c)(d+e) - bd - ce] + ce$
- This formula leads to $T(n) = 3T(n/2) + \theta(n)$
recurrence relation

- Using the master theorem this gives $\theta(n^{\log_2(3)})$ time complexity which is approximately $\theta(n^{1,585})$!

- It is used in general in optimization problems
- In such problems there are many possible solutions and we wish to find to one with the optimal value
- Solves problems by dividing them into smaller problems
- Uses the solutions from smaller problems to solve the bigger problem

- It has two main differences from divide and conquer:
 - Optimal substructure
 - Subproblems overlap
- Optimal substructure
 - A problem can be solved by dynamic programming only if the optimal solution of the problem you solve contains within it optimal solutions to the subproblems used for solving it

- We must ensure that the range of subproblems we consider includes those used for solving the initial problem
- So, it is essential that we are able to characterize the structure of the optimal solution of the problem by using the structure of the optimal solutions of the subproblems used for solving it
- Also, we must define the value for the optimal solution
- This is usually done by using a recursive formula

- The second main difference is that the problems overlap
- This can be solved by using a lookup table where we keep the values for the optimal solutions of the subproblems that we have solved so far
- The values for the optimal solutions for the subproblems can be solved in two ways:
 - In a bottom-up fashion in which we start from the smallest subproblems and build up to the initial problem

- The second way is to do it recursively by directly using the recurrence relation and stopping the recursion when we encounter a subproblem that we have already solved by using the lookup table
- This is called in literature memoization
- To tell if a subproblem has been already solved or not the lookup table will initially contain a special value that tells this

- In general the most difficult part in solving a problem by dynamic programming is to find the optimal substructure
- In practice this means finding the state that characterizes our problem and how to transition from the state of the initial problem to the states that characterize the subproblems used for solving the initial problem

- Problems
- Optimal matrix chain multiplication
- We are given the matrices $A_1 A_2 \dots A_n$ and we want to fully parenthesize the product of these matrices with the goal to perform a minimum number of scalar multiplications
- A matrix chain is fully parenthesized if it contains a single matrix or it contains two fully parenthesized matrices that are surrounded by an opening and a closing parenthesis

- The number of rows and columns for a matrix A_i is p_{i-1} and p_i
- This ensures that all the matrices can be multiplied together because they follow the rule that when multiplying two matrices the number of columns of the first must be equal to the number of lines of the second
- For matrices A_1, A_2, A_3 a full paranthesation is:
 - $(A_1(A_2A_3))$

- If matrix A_1 has the number of rows p and the number of columns q and matrix A_2 has the number of rows q and the number of columns r then the total number of scalar products when performing multiplication will be pqr
- Now that we have defined the problem, the first and most difficult step is to see if this problem has the optimal substructure property and to find it

- In order to do this we need to find out what subproblems we need to solve in order to ensure the optimal substructure property
- Let's say that we want to optimally paranthesize the sequence $A_i A_{i+1} \dots A_j$
- Let's consider that in the optimal solution to this problem we need to split the sequence in two at position k

- Now we have broken our problem in two subproblems
 - Paranthesizing the sequence $A_i A_{i+1} \dots A_k$
 - Paranthesizing the sequence $A_{k+1} A_{k+2} \dots A_j$
- Because we know that in order to achieve the optimal solution for the initial problem we must split the sequence at position k then the solutions for the above two subproblems must be optimal
- The proof is by contradiction

- Suppose that the paranthesation for the sequence $A_i A_{i+1} \dots A_k$ is not optimal
- Then it means that there exists a solution with less cost for the above sequence which also leads to a better solution for the initial problem
- Contradiction because we assumed that the optimal solution for the initial problem is achieved by splitting the sequence at position k

- In order to find the optimal solution we must consider all the possible splitting positions in order to not miss the one that leads us to the optimal solution
- Now we need to define the recurrence relation for computing the optimal values
- This comes from the optimal substructure property of dynamic programming

- Let $m[i,j]$ be the minimum number of scalar products for the sequence $A_i A_{i+1} \dots A_j$
- This can be defined using the following recurrence relation:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j , \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j . \end{cases}$$


- The simplest way to compute the solution is to use memoization
- If we want also to build the actual paranthesation we need to keep an aditional lookup table $s[i,j]$ that holds the split value at which we split in order to achieve the optimal solution for the sequence $A_i A_{i+1} \dots A_j$

- The paranthesation can be constructed using the following procedure:

```
PRINT-OPTIMAL-PARENS( $s, i, j$ )  
1  if  $i == j$   
2    print " $A$ " $i$   
3  else print "("  
4    PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )  
5    PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )  
6    print ")"
```

- Problem nr 2
- Rod-cutting problem. Given a rod of length n inches and a table of prices p_i with $i=1,2,\dots, n$ determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Also, each cut comes with the fixed cost c . If the price p_n for a rod of length n is large enough an optimal solution may require no cutting at all.

Solution

-  First, we reduce the problem to a simpler one by eliminating the fixed cost c . We can do this by subtracting from each price p_i the cost c . Now we only have to find out the maximum revenue without taking into account the additional cost c per cut. After we find the maximum revenue for our simplified problem we will add to it c because when we cut a rod in x pieces only $x-1$ cuts are done.

- The next step is to find out what subproblems should we try to solve. For this problem the subproblems that we want to solve is the optimal way to cut a rod of an arbitrary length x less than or equal to the initial length n . For a rod of length x we can initially cut out a rod of length i and then solve the subproblem corresponding to a rod of length $x-i$. It is easy to see now that we have found the structure of our optimal solution and that the optimal substructure property is respected.

- Let r_n be the maximum sum that can be obtained by cutting a rod of length n . The recurrence relation is straightforward:
- $r_0 = 0$
- $r_n = \max(p[i] + r_{n-i})$, with $i=1 \dots n$
- The time complexity is $O(n^2)$ because we have n problems and for a problem of size x we transition to x smaller problems

- If we implement the formula directly using recursion then we must use memoization
- Otherwise we will have exponential time complexity !
- More exactly 2^n where n is the initial length of the rod
- But it can also be implemented bottom-up pretty easy

- Problem nr 3
- Maximum subarray problem
- Given an array of length n find a subsequence whose sum is maximum.
- We have solved this problem previously using divide and conquer but the time complexity was $\Theta(n \log n)$
- Let's see if we can improve this using dynamic programming

- The major step is to find what subproblems should we solve optimally.
- We can take advantage of the fact that all the elements must be contiguous
- When we reach element at position i we have two choices:
 - Link it with the previous link
 - Start a new sequence from position i

- From this we can define the following subproblem:
 - What is the maximum sum we can obtain by building a subsequence that ends at position i , i included here
 - It is easy to see that this structure follows the optimal substructure rule

- If in order to obtain the optimal solution for position i we must link the current element with the previous one then the solution for the subproblem corresponding to position $i-1$ must be optimal, otherwise we would obtain a better solution for position i contradicting the fact that the solution for position i is optimal

• Let dyn_i be the sum of the optimal subsequence that ends at position i

• By optimal we mean the one with the best sum

- Then we can define dyn_i recursively using the following recurrence
 - $\text{dyn}_0 = v[0]$
 - $\text{dyn}_i = \max(\text{dyn}_{i-1} + v[i], v[i])$
- As we can see all the values can be computed in $\Theta(n)$
- The last thing that we must do is to update at each step the global maximum sum because the global optimal subsequence could end at any position !

- A nice observation is that we can implement this recurrence with constant memory because dyn_i only depends on dyn_{i-1} !
- Problems nr 4 and nr 5:
 - Given a string find the maximal length of a palindromic substring (by substring we mean that the elements it contains aren't necessary adjacent in the original string)
 - Given a $n \times n$ matrix find the sum of the submatrix with the maximum sum

- The elements of the submatrix must be adjacent in the original matrix