Time Complexity and the divide and conquer strategy

Or : how to measure algorithm run-time
And : design efficient algorithms

Oct. 2005

INP Grenoble
ENSIMAG

Contents

1. Initial considerations
   a) Complexity of an algorithm
   b) About complexity and order of magnitude
2. The “divide and conquer” strategy and its complexity
3. A word about “greedy algorithms”

Basic preliminary considerations

- We are interested by the *asymptotic time complexity* T(n) with n being the size of the input
- order of magnitude : $O(f(n))$
  - ➢ $\exists$ A, $\exists$ α $\forall$n>A $g(n) < \alpha f(n)$ => g is said to be $O(f(n))$
  - ➢ Examples :
    - n^2 is $O(n^3)$ (why?), $1000n + 10^{10}$ is $O(n)$

Understanding order of magnitude

If 1000 steps/sec, how large can a problem be in order to be solved in :

Time complexity	1 sec	1 min	1 day
$\log_2 n$	2^{1000}	∞	∞
n	1000	60 000	$8{,}6 \, . \, 10^7$
$n \log n$	140	4893	$5{,}6 \, . \, 10^5$
n^2	31	244	9300
n^3	10	39	442
2^n	10	15	26

Is it worth to improve the code?

- If moving from n^2 to $n.\log n$, definitively
- If your step is running 10 times faster,
  - ➢ For the same problem, 10 time faster!
  - ➢ For the same time how larger might the data be:
    - Linear : 10 time larger
    - $n.\log n$: almost 10 time larger
    - n^2 : 3 time larger
    - 2^n : initial size + 3.3

◄**Forget about**

Complexity of an algorithm

- Depends on the data :
  - If an array is already sorted, some sorting algorithms have a behavior in $O(n)$,
- Default definition : complexity is the complexity in *the worst case*
- Alternative :
  - Complexity in the best case (no interest)
  - Complexity on the average :
    - Requires to define the distribution of the data.

Complexity of a problem

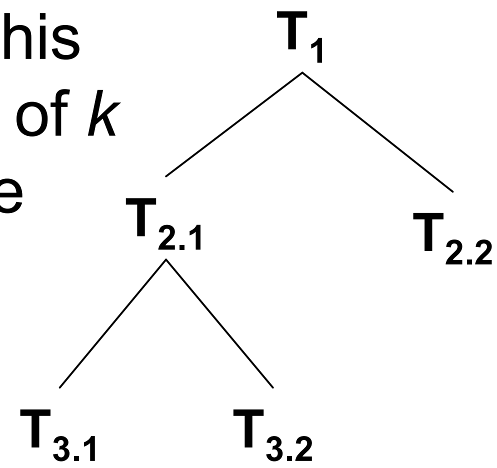
- The complexity of the best algorithm for providing the solution
  - Often the complexity is linear: you need to input the data;
  - Not always the case : the dichotomy search is in $O(n \log n)$ if the data are already in memory
- Make sense only if the problem can be solved :
  - Unsolvable problem : for instance: deciding if a program will stop (linked to what is mathematically undecidable)
  - Solvable problem: for instance: deciding if the maximum of a list of number is positive; complexity $O(n)$

Complexity of sorting

- Finding the space of solutions : one of the permutations that will provide the result sorted : size of the space : n!
- How to limit the search solution
  - Each answer to a test on the data specifies a subset of possible solutions
  - In the best case, the set of possible solution in cut into 2 half

Sorting (cont.)

- If we are smart enough for having this kind of tests : we need a sequence of *k* tests to reach a subset with a single solution.



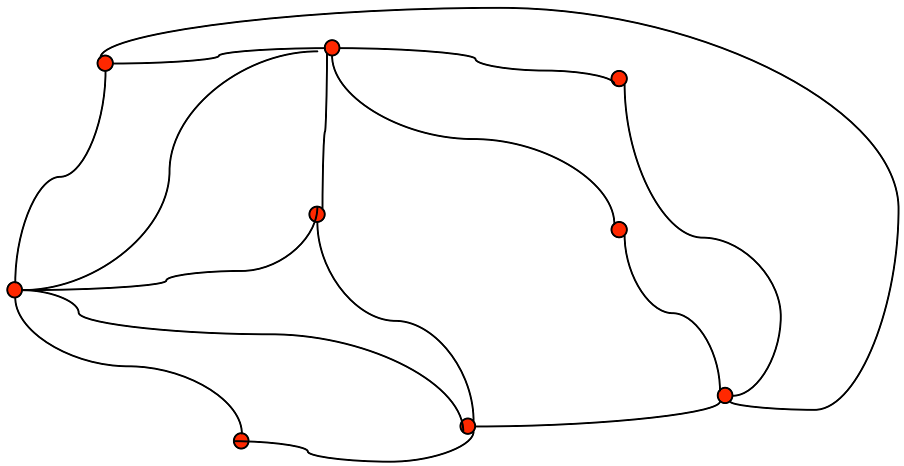
- Therefore : $2^k \sim n!$
- So

$$k \approx \log_2 !n \approx \log_2 \sqrt{2\pi n}\,\frac{n^n}{e^n} \approx \log_2 \sqrt{2\pi n} + n\log_2 n - n\log_2 e$$

- Therefore sorting is at best in $O(n.\log n)$
- And we know an algorithm in $O(n\log n)$

Examples of complexity

- Polyomial sum : $O(n)$
- Product of polynoms : $\cancel{O(n^2)}$? $O(n\log n)$
- Graph coloring : probably $O(2^n)$



- Are 3 colors for a planar graph sufficient?
- Can a set of numbers be splitted in 2 subsets of equal sum?

Space complexity

- Complexity in space : how much space is required?
  - ➢don't forget the stack when recursive calls occur
  - ➢Usually much easier than time complexity

The divide and conquer strategy

- A first example : sorting a set *S* of values
  - sort (*S*) =

```
if |S| ≤ 1 then return S
else divide (S, S1, S2)
     fusion (sort (S1), sort (S2))
end if
```

fusion is linear is the size of its parameter;
divide is either in O(1) or O(*n*)

The result is in *O*(*n*log*n*)

The divide and conquer principle

- General principle :
  - Take a problem of size *n*
  - Divide it into *a* sub problems of size *n/b*
  - this process adds some linear complexity *cn*
- What is the resulting complexity?

$$T(n) = aT\left(\frac{n}{b}\right) + cn$$

$$T(1) = 1$$

- Example . Sorting with fusion ; a=2, b=2

Fundamental complexity result for the divide and conquer strategy

- If $T(n) = aT(\frac{n}{b}) + cn$

  $T(1) = 1$

- Then
  - If a=b : $T(n) = O(n.\log n)$ ◄ Most frequent case
  - If a<b and c>0 : $T(n) = O(n)$
  - If a<b and c=0 : $T(n) = O(\log n)$
  - If a>b :

    $$T(n) = O(n^{\log_b a})$$

Proof : see lecture notes section 12.1.2

Proof steps

- Consider $n = b^k$ $(k = \log_b n)$
- $$\begin{aligned} T(n) &= aT(\frac{n}{b}) + cn \\ aT(\frac{n}{b}) &= a^2T(\frac{n}{b^2}) + a\frac{cn}{b} \\ \ldots & \quad \ldots \\ a^iT(\frac{n}{b^i}) &= a^{i-1}T(\frac{n}{b^{i+1}}) + a^i\frac{cn}{b^i} \\ a^{\log_b(n)}T(1) &= a^{\log_b(n)} \end{aligned}$$
- Summing terms together :

$$T(n) = cn\sum_{i=1}^{k-1}(\frac{a}{b})^i + a^k$$

Proof steps (cont.)

$$T(n) = cn\sum_{i=1}^{k-1}(\frac{a}{b})^i + a^k$$

- a<b ➔ the sum is bounded by a constant and $a^k < n$, so $T(n) = O(n)$
- a=b, c>0 ➔ $a^k = n$, so $T(n) = O(n.\log n)$
- a>b : the (geometric) sum is of order a^k/n
  - Both terms in a^k
  - Therefore $T(n) = O(n^{\log_b a})$

INP Grenoble ENSIMAG

Application: matrix multiplication

- Standard algorithm
  - For all (i,j) $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$ O(n³)
- Divide and conquer:
  - Direct way :

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{12} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A2 \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Counting : b=2, a=8

therefore $O(n^3)$!!!

- Smart implementation: Strassen, able to bring it down to 7
  - Therefore $O(n^{\log_2 7}) = O(n^{2,81})$

Only for large value of *n* (>700)

INP Grenoble ENSIMAG

Greedy algorithms : why looking for

- A standard optimal search algorithm:

Computes the best solution extending a partial solution S' only if its value exceeds the initial value of *Optimal_Value;*

The result in such a case is *Optimal_S;* these global variables might be modified otherwise

```
Search (S: partial_solution) :
    if Final(S) then if value(S)> Optimal_Value then
                            Optimal_Value := value(S); Optimal_S := S;
                        end if;
    else for each S' extending S loop
                Search (S');
    end if
```

Complexity : if k steps in the loop, if the search depth is n : $O(k^n)$

Instantiation for the search of the longest path in a graph

```
Longest (p: path)
-- compute the longest path without circuits in a graph
--      only if the length extends the value of The_Longest set
--      before the call; in this case Long_Path  is the value of this path, …..
if Cannot_Extend(p) and then length(p)> The_Longest
            then The_Longest := length(p); Long_Path := p;
 else let x be the end node of p;
        for each edge (x,y) such that y ∉ p loop
            Longest (p ⊕ y);
end if;
-- initial call : The_Longest := -1;
              Longest (path (departure_node));
```

Alternative

- Instead of the best solution, a *not too bad* solution?

```
Greedy_search(S: partial_solution) :
  if final (S) then sub_opt_solution := S
  else select the best S' expending S
       greedy_search (S')
  end if;
```

Complexity : $O(n)$

Greedy search for the longest path

```
Greedy_Longest (p: path) :
  if Cannot_Extend(p) then Sub_Opt_Path := p
  else let x be the end node of p;
       select the longest edge (x,y) such that y ∉ p
  exp
       Greedy_Longest (p ⊕ y);
  end if;
```

Obviously don't lead to the optimal solution in the general case

Exercise : build an example where it leads to the worst solution.

How good (bad?) is such a search?

- Depends on the problem
  - Can lead to the worst solution in some cases
  - Sometimes can guarantee the best solution

Example : the minimum spanning tree (find a subset of edges of total minimum cost connecting a graph)

```
Edge_set := ∅
for i in 1..n-1 loop
        Select the edge e with lowest cost not connecting already
        connected nodes
        Add e to Edge_set
    End loop;
```

- Notice that this algorithm might not be in $O(n)$ as we need to find a minimum cost edge, and make sure that it don't connect already connected nodes
  - This can be achieved in log$n$ steps, but is out of scope of this lecture : see the "union-find" data structure in Aho-Hopcroft-Ulman

Conclusion: What to remember

- Complexity on average might differ from worst case complexity : smart analysis required
- For unknown problems, explore first the size of solution space
- *Divide and conquer* is an efficient strategy (exercises will follow); knowing the complexity theorem is required
- Smart algorithm design is essential: a computer 100 times faster will never defeat an exponential complexity