Algorithms and Data Structures in Biology

Divide and Conquer Algorithms

Ugo Dal Lago

University of Bologna, Academic Year 2018/2019

The Divide and Conquer Approach

- In the divide and conquer approach to algorithm design, one:
  - First **partitions** the underlying problem instance to two, "smaller", instances.
  - Then **solves** them separately.
  - Finally **aggregates** the results.
- This way of proceeding often leads to fast algorithms, or to an improvement over existing algorithmic techniques.

An Efficient Sorting Algorithm

```
MergeSort(c)
n ← size of c
if n = 1
    return c
left ← list of first n/2 elements of c
right ← list of last n − n/2 elements of c
sortedLeft ← MergeSort(left)
sortedRight ← MergeSort(right)
sortedList ← Merge(sortedLeft, sortedRight)
return sortedList
```

The Merge Routine

MERGE($\mathbf{a}, \mathbf{b}$)

$n1 \leftarrow$ size of $\mathbf{a}$
$n2 \leftarrow$ size of $\mathbf{b}$
$a_{n1+1} \leftarrow \infty$
$b_{n2+1} \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ **to** $n1 + n2$
    if $a_i < b_j$
        $c_k \leftarrow a_i$
        $i \leftarrow i + 1$
    else
        $c_k \leftarrow b_j$
        $j \leftarrow j + 1$
return $\mathbf{c}$

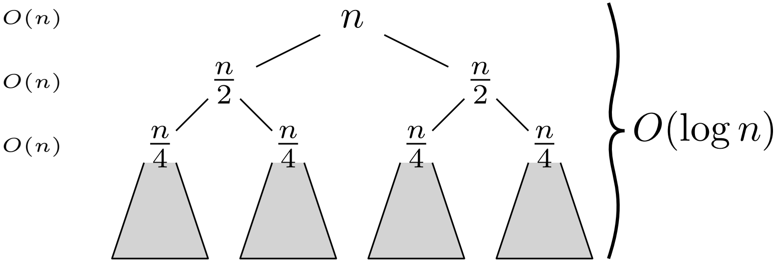
Analysing Merge Sort Runtime

$$T(n) = 2T(n/2) + cn$$
$$T(1) = 1$$

- All in all, the time complexity is thus $O(n \log n)$.

Analysing Merge Sort Runtime

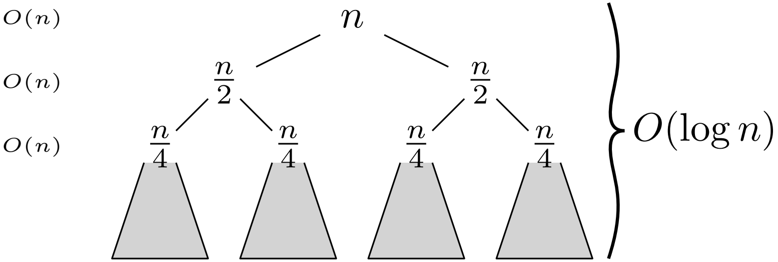
$$T(n) = 2T(n/2) + cn$$
$$T(1) = 1$$



- All in all, the time complexity is thus $O(n \log n)$.

Analysing Merge Sort Runtime

$$T(n) = 2T(n/2) + cn$$
$$T(1) = 1$$



- All in all, the time complexity is thus $O(n \log n)$.

A Closer Look at the Complexity of Global Alignment

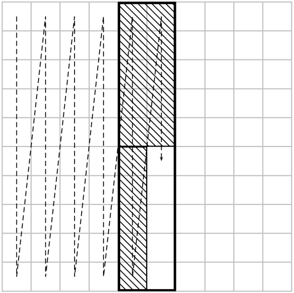
- We already know that the the Global Alignment problem can be effectively solved by way of **dynamic programming**.
  - One just to have to visit all nodes of the edit graph in an appropriate order.
  - For each node in the edit graph, just a constant amount of work has to be done.
- If n and m are the length of the two strings involved, the *time complexity* is easily seen to be $O(nm)$.
- But how about the *space complexity*?
  - The algorithm space consumption is itself $O(nm)$.
  - For each node of the edit graph (i, j), one should keep track of the value $s_{i,j}$.

Subquadratic Space Complexity?

- Is it necessary to keep track of the *entire* matrix $s_{i,j}$?
- If we are only interested in the *score* of the optimal alignment, we can just keep track, e.g., of the *last column*.
- But what if we are interested in computing *the alignment itself*, namely the path in the edit graph having maximal score?

Subquadratic Space Complexity?

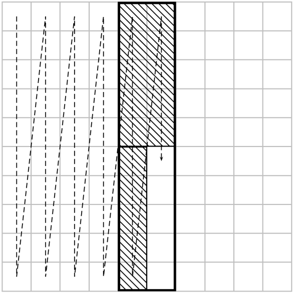
- Is it necessary to keep track of the *entire* matrix $s_{i,j}$?
- If we are only interested in the *score* of the optimal alignment, we can just keep track, e.g., of the *last column*.



- But what if we are interested in computing *the alignment itself*, namely the path in the edit graph having maximal score?

Subquadratic Space Complexity?

- Is it necessary to keep track of the *entire* matrix $s_{i,j}$?
- If we are only interested in the *score* of the optimal alignment, we can just keep track, e.g., of the *last column*.



- But what if we are interested in computing *the alignment itself*, namely the path in the edit graph having maximal score?

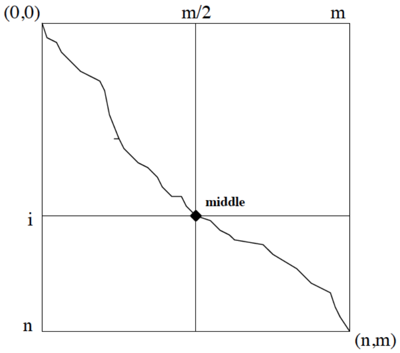
Subquadratic Space Complexity?

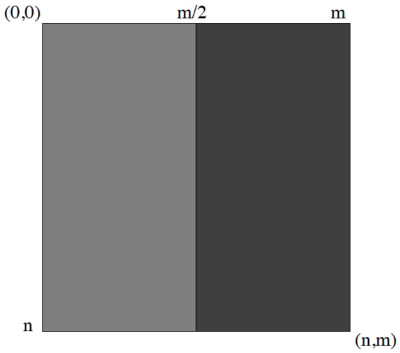
- If we also want to compute the path, and not just its score, divide and conquer can come to the resque.
- We can reason as follows:
  - First of all, focus on the middle column.
  - Compute the maximal scores of the nodes in the middle column in the edit graph.
  - Compute the maximal scores of the nodes in the middle column in the *reversed* edit graph
  - An optimal path can be found through the node with coordinates $(i, \frac{m}{2})$ such that the sum of its two scores is maximal.
  - Then, look for an optimal path from the source to $(i, \frac{m}{2})$, for an optimal path from $(i, \frac{m}{2})$ to the target.

The Algorithm

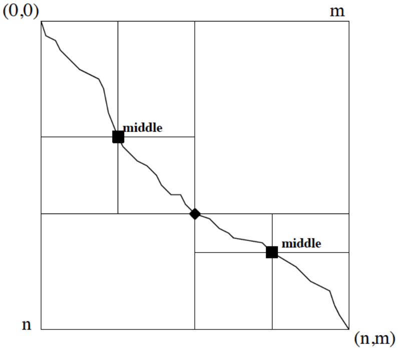
```
PATH(source, sink)
if source and sink are in consecutive columns
        output longest path from source to sink
else
        mid ← middle vertex (i, m/2) with largest score length(i)
        PATH(source, mid)
        PATH(mid, sink)
```

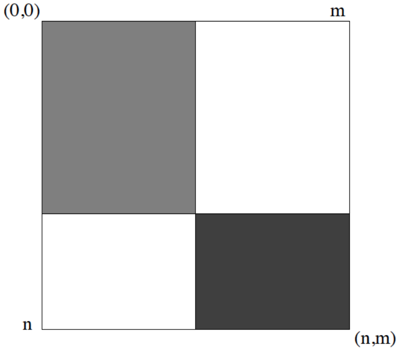
The Complexity of PATH



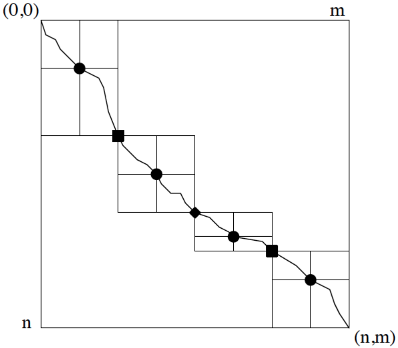


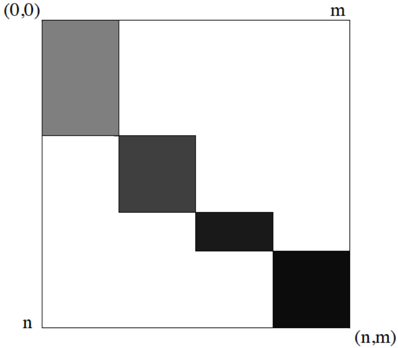
The Complexity of Path





The Complexity of Path





The Complexity of PATH

- **Time Complexity**
  - The total area of the visited rectangle is, roughly, the complexity of the algorithm.
  - The complexity is thus proportional to:

$$a + \frac{a}{2} + \frac{a}{4} + \ldots = a(1 + \frac{1}{2} + \frac{1}{4} + \ldots) = 2a$$

    where a is the area of the whole rectangle, namely $O(nm)$.
- **Space Complexity**
  - Of course we need to compute some s_{ij}, many of them repeatedly.
  - At any moment in time, however, we need to keep track of just *a linear number of them*.
  - The space complexity is thus $O(\max\{m, n\})$.

Block Alignments

- Is it possible to go beyond $O(n^2)$ when looking for efficient algorithms for the global alignment problem of two strings of equal length n?
  - This is an extremely interesting, but still open, research problem.
- Something can be definitely be said when the input strings are divided into *blocks*.
  - A string $\mathbf{u}$ is a *t-block* string if there is n such that

    $$\mathbf{u} = u_1 \cdots u_n$$

    and t divides n. A t-block strings can be seen as being naturally divided into $\frac{n}{t}$ blocks of length t
  - A block alignment of two t-block strings $\mathbf{u}$ and $\mathbf{v}$ is an alignment in which every block in one sequence is aligned against a whole block with the other sequence, or inserted or deleted *as a whole*.

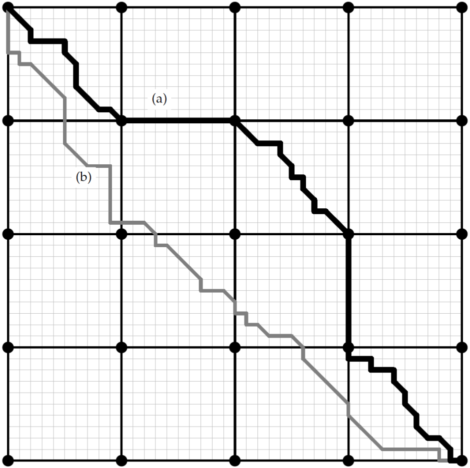
Block Alignments

- Is it possible to go beyond $O(n^2)$ when looking for efficient algorithms for the global alignment problem of two strings of equal length n?
  - This is an extremely interesting, but still open, research problem.
- Something can be definitely be said when the input strings are divided into *blocks*.
  - A string $\mathbf{u}$ is a t-*block* string if there is n such that

    $$\mathbf{u} = u_1 \cdots u_n$$

    and t divides n. A t-block strings can be seen as being naturally divided into $\frac{n}{t}$ blocks of length t
  - A block alignment of two t-block strings $\mathbf{u}$ and $\mathbf{v}$ is an alignment in which every block in one sequence is aligned against a whole block with the other sequence, or inserted or deleted *as a whole*.

Block Alignments



The Block Alignment Problem

Block Alignment Problem:
Find the longest block path through an edit graph.

Input: Two sequences, **u** and **v** partitioned into blocks of size t.

Output: The block alignment of **u** and **v** with the maximum score (i.e., the longest block path through the edit graph).

A Simple Algorithmic Solution

- One can consider each $t \times t$ block separately, and for each of them solve the global alignment problem.
  - Each of these mini-alignment problems can be solved in time $O(t^2)$.
  - Since, altogether, there are $\frac{n}{t} \cdot \frac{n}{t}$, the overall complexity is of course
$$\frac{n}{t} \cdot \frac{n}{t} \cdot O(t^2) = O\left(\frac{n^2 \cdot t^2}{t^2}\right) = O(n^2).$$
  - This way, we can compute the score $\beta_{i,j}$ between the i-th block of $\mathbf{u}$ and the j-th block of $\mathbf{v}$
- Then, the results of the previous step can be aggregated on block basis, by way of the following recurrence:
$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma_{block} \\ s_{i,j-1} - \sigma_{block} \\ s_{i-1,j-1} + \beta_{i-1,j-1} \end{cases}$$
where σ_{block} is the penalty for inserting or deleting an entire block.
  - This second step has of course complexity $O(\frac{n^2}{t^2})$.

A Simple Algorithmic Solution

- One can consider each $t \times t$ block separately, and for each of them solve the global alignment problem.
  - Each of these mini-alignment problems can be solved in time $O(t^2)$.
  - Since, altogether, there are $\frac{n}{t} \cdot \frac{n}{t}$, the overall complexity is of course
$$\frac{n}{t} \cdot \frac{n}{t} \cdot O(t^2) = O\left(\frac{n^2 \cdot t^2}{t^2}\right) = O(n^2).$$
  - This way, we can compute the score $\beta_{i,j}$ between the i-th block of $\mathbf{u}$ and the j-th block of $\mathbf{v}$
- Then, the results of the previous step can be aggregated on block basis, by way of the following recurrence:
$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma_{block} \\ s_{i,j-1} - \sigma_{block} \\ s_{i-1,j-1} + \beta_{i-1,j-1} \end{cases}$$
  where σ_{block} is the penalty for inserting or deleting an entire block.
  - This second step has of course complexity $O(\frac{n^2}{t^2})$.

So What?

- The overall complexity of the just-sketched algorithm is thus dominated by the first step, which takes $O(n^2)$ time.
  - This is *no better* than the complexity of the usual dynamic programming algorithm.
  - This is unsurprising, but remarkable, because we are solving a different problem anyway.
- In some cases, it makes sense to modify the algorithm in its first part.
  - Instead of solving $\frac{n^2}{t^2}$ mini-alignment problems, one for each block, we solve **all possible** mini-alignment problems about strings of length t.
  - If the underlying alphabet is $\{A, T, C, G\}$, then there are $4^t \cdot 4^t$ such problems.
  - For certain values of t, this can makes a lot of sense.

So What?

- The overall complexity of the just-sketched algorithm is thus dominated by the first step, which takes $O(n^2)$ time.
  - This is *no better* than the complexity of the usual dynamic programming algorithm.
  - This is unsurprising, but remarkable, because we are solving a different problem anyway.
- In some cases, it makes sense to modify the algorithm in its first part.
  - Instead of solving $\frac{n^2}{t^2}$ mini-alignment problems, one for each block, we solve **all possible** mini-alignment problems about strings of length t.
  - If the underlying alphabet is $\{A, T, C, G\}$, then there are $4^t \cdot 4^t$ such problems.
  - For certain values of t, this can makes a lot of sense.

So What?

If $t = \frac{\log_2 n}{4}$, then:

- The **first step** of the algorithm would take time

$$\begin{aligned} 4^t \cdot 4^t \cdot O(t^2) &= 4^{\frac{\log_2 n}{4}} \cdot 4^{\frac{\log_2 n}{4}} \cdot O(\log^2 n) \\ &= (2^{\log_2 n})^{\frac{1}{2}} \cdot (2^{\log_2 n})^{\frac{1}{2}} \cdot O(\log^2 n) \\ &= n^{\frac{1}{2}} \cdot n^{\frac{1}{2}} \cdot O(\log^2 n) = O(n \log^2 n) \end{aligned}$$

- The **second step** of the algorithm would instead take time

$$O\left(\frac{n^2}{t^2}\right) \cdot O(\log n) = O\left(\frac{n^2}{\log n}\right).$$

- Overall, the complexity is dominated by the second step, thus being $O\left(\frac{n^2}{\log n}\right)$.

Thank You!

Questions?