3 Algorithm Techniques

In this chapter we will briefly review some of the algorithmic techniques commonly used in bioinformatics.

3.1 Algorithms

An *algorithm* is a well-defined sequence of steps used to solve a well-defined problem in finite time.

An algorithm must solve all instances of the problem for which it was designed and thus is said to be *correct*.

The *running time* of an algorithm is the number of instructions it executes when run on a particular instance.

For the analysis of the algorithm the running time is often computed for a *worst case* instance of the problem, or sometimes for the *average case*, if the distribution of input instances is known.

Conceptually we distinguish between

- algorithm strategy
- algorithm structure
  - recursive
  - iterative
- algorithm solution
  - find a good solution
  - find best(s) solution(s)

3.2 Algorithm strategies

There are a number of well-known algorithm strategies:

- Recursive algorithms
- Backtracking algorithms
- Branch and bound algorithms
- Divide and conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Brute force algorithms
- Heuristic algorithms

3.3 Recursion

A combinatorial problem: Fibonacci numbers

n	0	1	2	3	4	5	6	7	8	9	10	11
F_n	0	1	1	2	3	5	8	13	21	34	55	89

The problem of the Fibonacci numbers is a classic example of a recursion problem:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

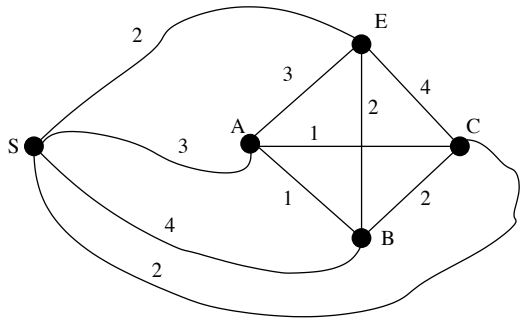
[fragile] Recursions: reapply algorithm to subproblem

Another example: $N!$, the factorial of a number N:

```
function fact(N){
if(N==1)
        return 1
else
        return N*fact(N-1)
}
```

3.4 Exhaustive search and backtracking

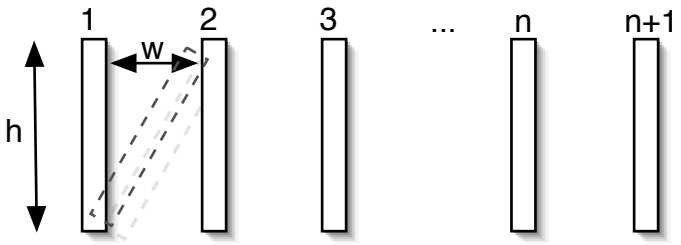
The solution to combinatorial problems often requires an exhaustive search of the set of all possible solutions.
Example: The Travelling Salesman Problem
The traveling salesman problem (TSP) asks for the shortest route to visit a collection of cities and return to the starting point.



Backtracking is a general technique for organizing the exhaustive search for a solution to a combinatorial problem.

The backtracking technique can be applied to those problems that exhibit the *domino principle*: if a constraint is not satisfied by a partial solution, the constraint will not be satisfied by any extension of the partial solution to a global solution.

Domino principle



Given h (height of a domino) $> w$ (space in between dominos):
we knock over the first domino
if nth domino falls, then $(n+1)$st domino will fall.

Many times a problem may be expressed in terms of detecting a particular class of subgraph in a graph. Then the backtracking approach to solving such a problem would be:
Scan each node of the graph, following a specific order, until
(1) a subgraph constituting a solution has been found or
(2) it is discovered that the subgraph built so far cannot be extended to be a solution.

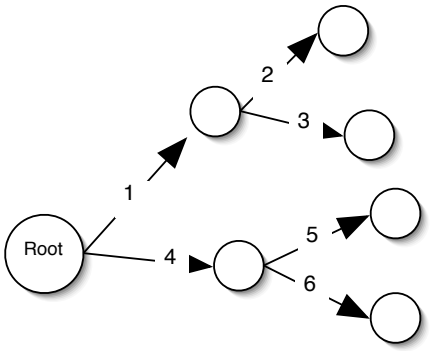
If (2) occurs then we go "back-up" the search process until a node is reached from which a solution might still be found.

3.5 Backtracking and Depth-first search

Assume that we can represent the space of solutions as a tree (or graph) and that a search for feasible solution is done on the tree (or graph).

Then backtracking can be performed in a depth-first search (DFS) manner by traversing the tree as follows:

Starting at the root, explore as far as possible along each path before backtracking.



3.6 Branch-and-Bound

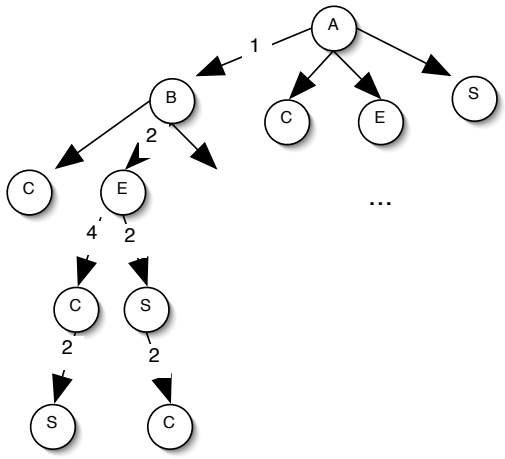
The branch-and-bound method can be used for finding one or all solutions of a combinatorial problem, where solutions are associated with a cost, such that the cost of the whole solution cannot be smaller than the cost of any partial solution.

The technique consists of remembering the lowest-cost solution found at each stage of the backtracking search, and to use the cost of the lowest-cost solution found so far as a lower bound on the cost of a least-cost solution to the problem, in order to discard partial solutions with costs larger than the lowest-cost solution found so far.

Represent the search for a solution as a tree, the so-called *branch-and-bound* tree.

The root of the tree is a dummy node of cost zero, the nodes at level one represent the possible values which the first variable can be assigned to, the nodes at level two represent the possible values which the second variable can be assigned to, given the value which the first variable was assigned to, and so on.

Subtrees in the tree rooted at nodes of cost greater than the cost of a previous leaf node, are pruned off the tree.



Note: Branch and bound does not decrease the complexity of a problem, it is simply a reasonable way to organize a search for an optimal solution.

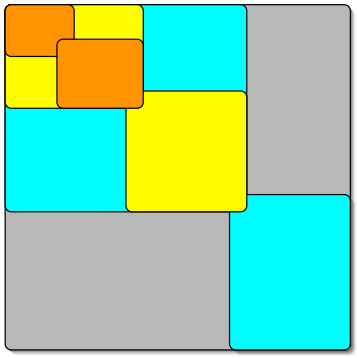
Example: The most parsimonious phylogenetic tree

3.7 Divide-and-Conquer

To solve a problem on an instance of size n, a solution is found either directly because solving that instance is easy (typically, because the instance is small) or the instance is divided into two or more smaller instances. Each of these smaller instances is recursively solved, and the solutions are combined to produce a solution for the original instance.

1. Given a problem, identify a small number of significantly smaller subproblems of the same type
2. Solve each subproblem recursively (the smallest possible size of a subproblem is a base-case)
3. Combine these solutions into a solution for the main problem

The name divide and conquer indicates that a problem is solved by first dividing it into smaller and easier problems and then conquering each subproblem separately.



The divide-and-conquer technique can be applied to those problems that exhibit the *independence principle*:

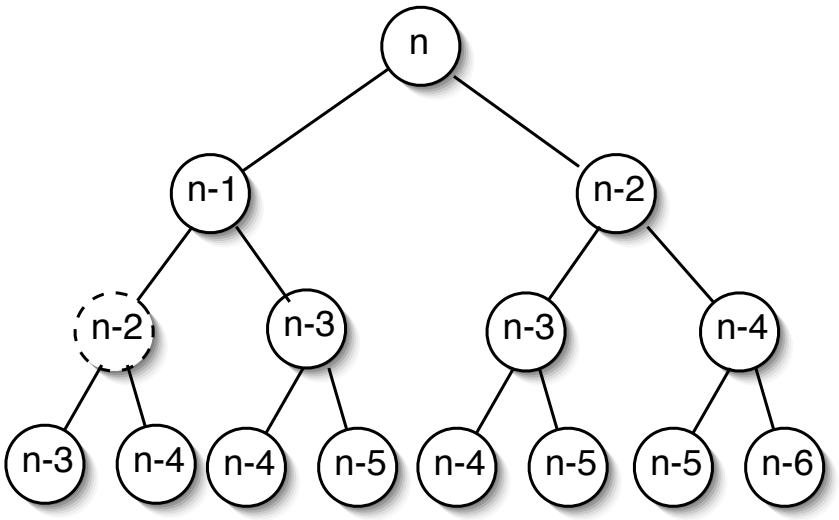
Each problem instance can be divided into a series of smaller problem instances which are independent of each other.

Example: One of the simplest examples is "MergeSort" of an array: Partition the array into two equal parts, sort each of the two parts using MergeSort and then merge the two sorted parts to sort the whole array.

[fragile] When a problem is solved by "divide-and-conquer", sometimes the same subproblem appears multiple times. A recursive algorithm for the divide-and-conquer according to this definition is:

```
Fibonacci-R(i){
if i = 0
      then return 0
else {
if i = 1
       then return 1
else return Fibonacci-R(i-1) + Fibonacci-R(i-2)}
}
```

However, it is easy to see that the algorithm is not efficient, since values of F_i are calculated several times independently.



3.8 Dynamic Programming

The "dynamic programming" paradigm is most often applied in the construction of algorithms to solve a certain class of *optimisation problems*, ie. problems which require the minimisation or maximisation of some measure.

It is applicable when a large search space can be structured into a succession of stages, such that the initial stage contains trivial solutions to sub-problems, each partial solution in a later stage can be calculated by recurring on only a fixed number of partial solutions in an earlier stage, the final stage contains the overall solution. The method usually accomplishes this by maintaining a table or matrix of sub-instance results.

Dynamic programming can be thought of as being the reverse of recursion or divide-and-conquer. Divide-and-conquer is a top-down mechanism – we take a problem, split it up, and solve the smaller

problems that are created. Dynamic programming is a bottom-up mechanism – we solve all possible small problems and then combine them to obtain solutions for bigger problems.

A general DP algorithm consists of 4 steps:

1. Characterisation of the structure of the (an) optimal solution
2. Recursive definition of the value of an optimal solution
3. Computation of a table of results, computing the solutions of larger subproblems from the solutions of smaller ones.
4. Construction of an optimal solution through the computed optimal value.

[fragile] The "dynamic programming" version of the algorithm for the computation of the Fibonacci numbers is:

```
Fibonacci-DP(i){
  A[0] <- 0        // assume 0 is a valid index
  A[1] <- 1
  for j <- 2 to i
      A[j] <- A[j-1] + A[j-2]
  return A[i]}
```

Here we see the basic elements of dynamic programming: though the solution is still defined in a "top-down" manner (from solution to sub-solution), it is built in a "bottom-up" manner (from sub-solution to solution). Since the solutions to subproblems are kept in a data structure for possible later use, each of them is only calculated once.

Another example: computation of the binomial coefficient using the formula

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Exercise: why is a divide-and-conquer approach computationally infeasible here?