UMSL

Analyzing algorithms, Growth of functions, and Divide-and-conquer

What kinds of problems are solved by algorithms?

Biological problems

- The human DNA contains approximately 3 billion of these base pairs and requires 1 GB of storage space
- http://sysbio.rnet.missouri.edu/chromosome3d/about.php
- Storing the the information is a challenge. Why?
- Processing each DNA sequence is a challenge. How?

Data travel routes in the Internet

- Internet enables people to quickly access and retrieve large amounts of data
- How can web-sites manage and manipulate large volume of data
- How to find the good routes on which data will travel?

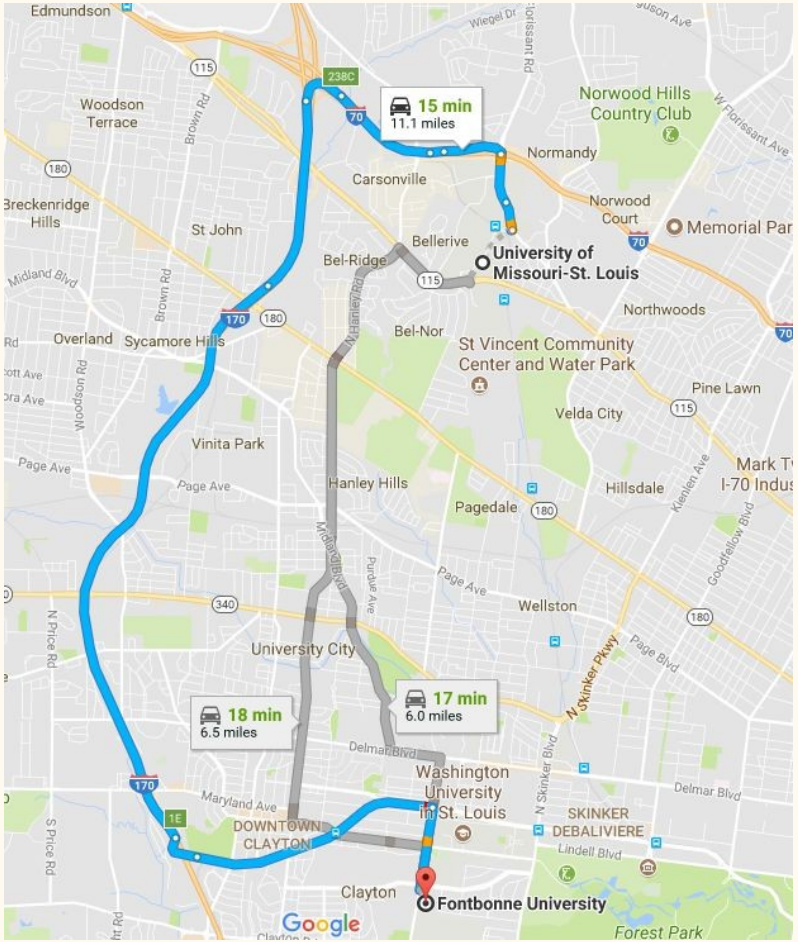
Analyzing data for winning an election

- A political candidate wants to determine where to spend money for advertising in order to maximize the chances of winning an election.
- Examples of input: population, type of area, building roads, tax, farm subsidies, etc.

Some specific problems

How to find shortest path between two cities?

What would you do if you did not have an algorithm for finding the shortest path?



Some specific problems

How to find the longest common subsequence?

$X = \langle x_1, x_2, \cdots, x_m \rangle$ and $Y = \langle y_1, y_2, \cdots, y_m \rangle$ are ordered sequence of symbols.

The length of the common subsequence of X and Y gives one measure of how similar the two sequences are.

Multiple sequence alignments examples

What would you do if you did not have an algorithm for the same?

Some specific problems

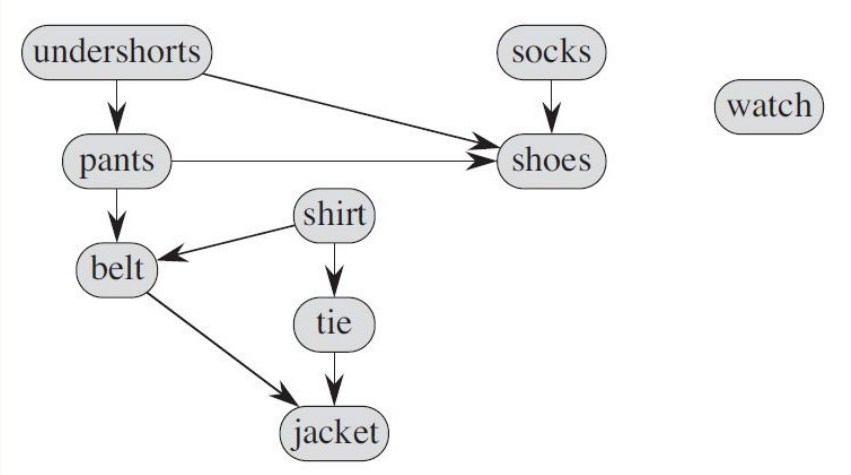
A assembler (person or a machine) receives parts of a machine to complete a mechanical design.

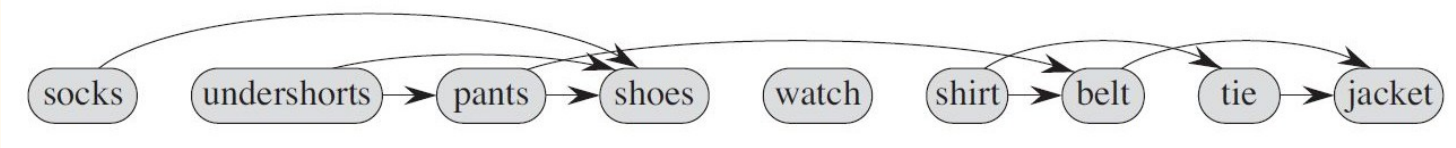
You know what parts you need and which parts depend on which.

In what order should you present the parts?

What would you do if you did not have an algorithm for the same?

Some specific problems





Two characteristics common to many of these algorithms

(a) They have many candidate solutions, but most of them are not the 'best'. Finding the best is challenging.

(b) The have practical applications.

Applications of shortest-path algorithms?

Applications of topological sorting?

Applications of shortest common subsequence?

An example of a Hard Problem

The Travelling Salesman Problem (TSP)

- Consider that FedEx has a central depot.
- Each day, the delivery truck loads at the depot and sends it around to deliver mails to several addresses.
- FedEx wants to select an order of delivery stops that yields the lowest overall distance travelled by the truck.

There is no known efficient algorithm for this problem.

NP-complete problems need ‘approximation algorithms’

Some definitions

Algorithm: any well-defined computational procedure that takes some value (or set of values) as input and produces a value (or set of values) as output. *Like a cooking recipe!*

Correct algorithm: an algorithm is said to be correct if, for every input instance, it halts with the correct output.

Data structure: a way to store and organize data in order to facilitate access and modifications. There is no single best data structure. Why?

Analyzing an algorithm: predicting the resources that the algorithm requires. For example, memory, computational time, communication bandwidth, etc. *Not checking whether it works or not!*

- Algorithms is a technology!

Analyzing an algorithm

A computer program may need to have a lot of features.

Some things to consider are: user-friendliness, robust, maintainable, less coding time, memory usage, bandwidth usage, etc.

But, most of the time, we are concerned with the **speed** or **running time**.

What is the best way to analyze the running time? *Supply a huge input!*

Asymptotic analysis is the tool we will use.

Designing algorithms

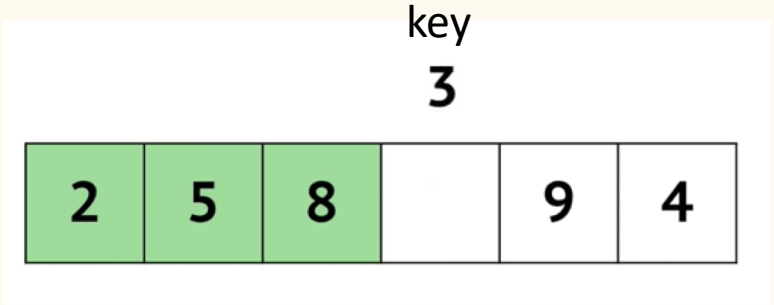
There are many algorithm design techniques - incremental, divide-and-conquer (recursive), dynamic programming, greedy, genetic, etc.

Example of incremental approach: **insertion sort**

Example of divide-and-conquer: **merge sort**

Insertion sort (Incremental approach)

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



Analysis of insertion sort

Total execution time = number of times the for loop is executed * the number of times the while loop is executed

The number of times the while loop is executed = 1 + 2 + 3 + ⋯+ (n-1)

Total number of computations = $a * n^2 + b * n + c$

Best case vs Worst case: already sorted *vs* already reverse sorted

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Divide-and-conquer approach

Many useful algorithms are recursive in structure.

This approach involves three steps at each level of recursion:

(a) **Divide** the problem into a number of subproblems that are smaller instances of the same problem.

(b) **Conquer** the subproblem by solving recursively; if the subproblem is small enough, solve it in a straightforward manner. *a lazy conqueror!*

(c) **Combine** the solutions to the subproblems into the solution for the original problem.

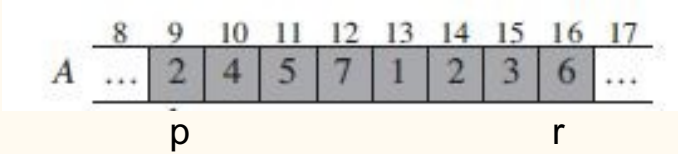
Merge sort

Operation of merge sort:

Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each

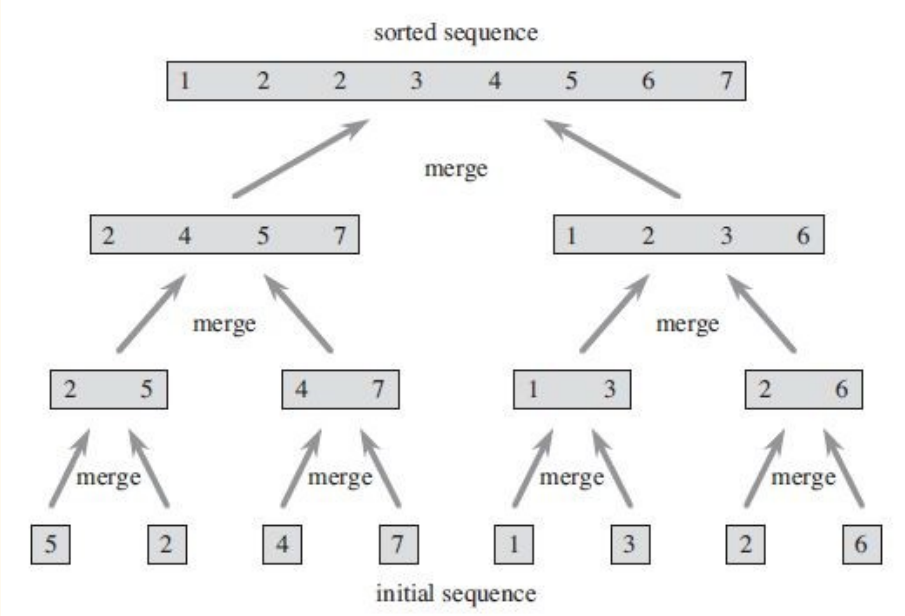
Conquer: Sort the two subsequences recursively *using merge sort.*

Combine: Merge the two sorted subsequences to produce sorted answer




```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Example of merge sort and recurrence equation



The operation of merge sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

T(n/b) is the time needed to solve the problem of size n/b

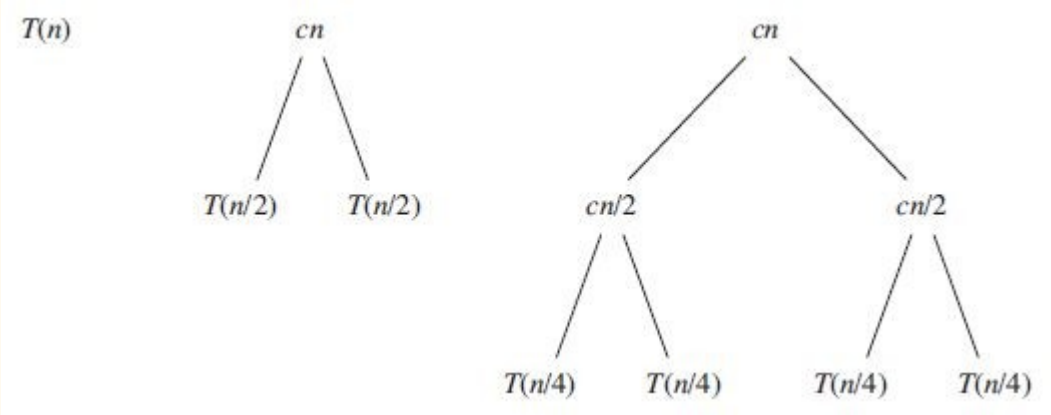
D(n) is the time needed for divide the problem into subproblems

C(n) is the time needed to combine solutions to the subproblems

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

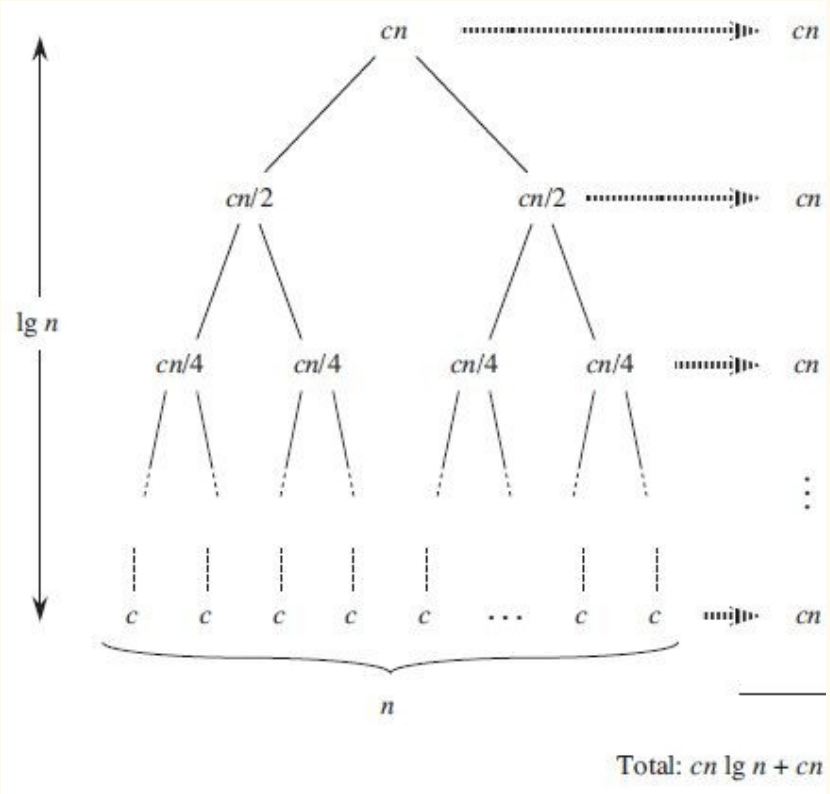
Recursion tree

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases}$$



Recursion tree

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1, \end{cases}$$



The height of the tree is lg n. lg (8) = 3.

Number of levels = lg(n) + 1.

Total cost = c n * (lg(n) + 1)

= c n lg(n) + d n

lg (n) stands for $\log_2(n)$

Are there best/worst caserunning times?

Order of growth

Cost of insertion sort = $a\,n^2 + b\,n + c$

Cost of merge sort = $c\,n\,\lg(n) + d\,n$

We are concerned about how the running time of algorithm increases with the size of the input, as the size of the input increases without bound.

i.e. we would like to study the **asymptotic efficiency** of algorithms

An algorithm that is asymptotically more efficient, will best choice unless when we have very small inputs

Asymptotic notations - big theta Θ

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

A function f(n) belongs to Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$.

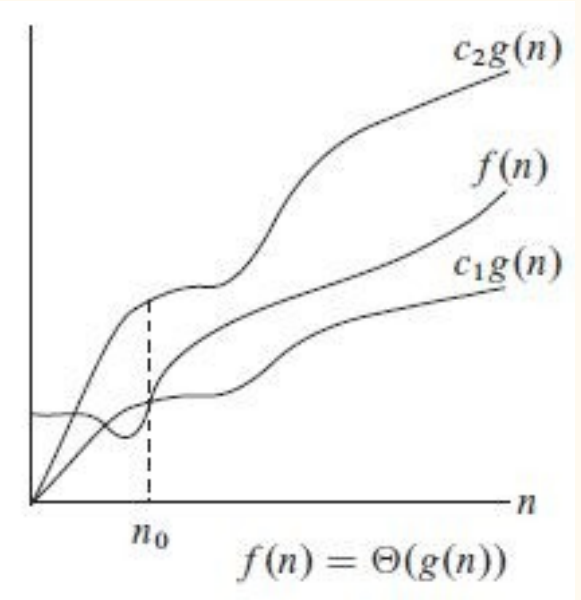
Example: Merge sort = > f(n) = c n lg(n) + d n

Running time = 1 * n * lg(n) to 1000000 * n * lg(n)

c n lg(n) + d n = Θ(**n lg(n)**)

Merge sort's running time is Θ(n lg(n))

We say that g(n) is an **asymptotically tight bound** for f(n).



Asymptotic notations - big oO

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

A function f(n) belongs to O(g(n)) if there exists a positive constant c such that it is less than c g(n).

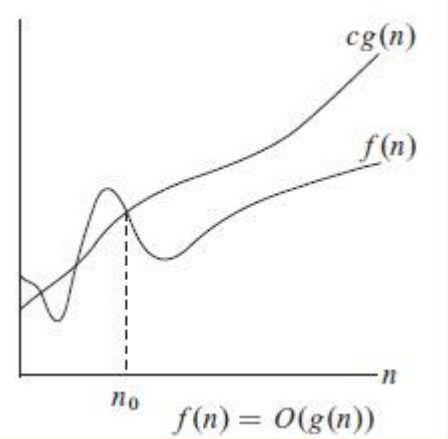
Example: Insertion sort = > f(n) = a * n^2 + b * n + c

Running time = 0 to 1000000 * n^2

a * n^2 + b * n + c = O(n^2)

Insertion sort' s running time is O(n^2).

O-notation provides an **asymptotic upper bound** for f(n).



Asymptotic notations - big omega Ω

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

A function f(n) belongs to Ω(g(n)) if there exists a positive constant c such that it is greater than c g(n).

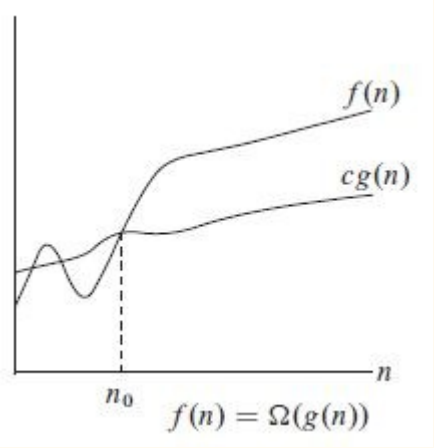
Example: Insertion sort (best case) = > f(n) = a * n + b * n + c

Running time = 0 to 1000000 * n

$a * n^2 + b * n + c = \Omega(n)$

Insertion sort's running time is Ω(n).

Ω-notation provides an **asymptotic lower bound** for f(n).



Which statement is wrong?

(a) Insertion sort's running time is $\Omega(n)$.
(b) Insertion sort's best-case running time is $\Omega(n)$.
(c) Insertion sort's running time is $O(n^2)$.
(d) Insertion sort's running time is $\Theta(n^2)$.
(e) Insertion sort's worst-case running time is $\Theta(n^2)$.
(f) Merge sort's running time is $O(n \lg(n))$.
(g) Merge sort's running time is $\Omega(n \lg(n))$.
(h) Merge sort's running time is $\Theta(n \lg(n))$.

Insertion sort – n to n^2
Merge sort – always - n * lg(n)

BigTheta – tight bound
BigO – upper bound
BigOmega – lower bound