Algorithms in Computational Biology

Shaojie Zhang

Computer Science

Contact Details:

Shaojie Zhang:

- Office: HEC 311
- Telephone: 407-823-6095
- Email: shzhang@cs.ucf.edu (No Webcourse Emails!!!)

Course home page: Webcourses@UCF

Misc.

- Time and Location
  - HEC-104
  - T/H 3:00-4:15 pm
  - Office hours: 2:00-2:50 pm

- Textbook (references):
  - Pavel Pevzner: Computational molecular biology: an algorithmic approach , MIT Press, 2000
  - Dan Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge university press, 1997
  - Phillip Compeau and Pavel Pevzner, Bioinformatics Algorithms: An Active Learning Approach, Volume 1 and Volume 2. Active Learning Publishers, 2015

Assessment:

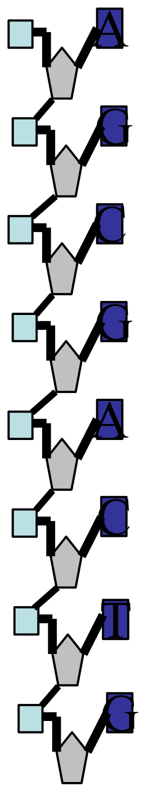
Element	Weight
4 Assignments: only accept answers in tex file (**Latex)**	40%
Midterm	20%
Final Exam	20%
Algorithm Presentation	15%
Class Participation	5%
TOTAL	100%

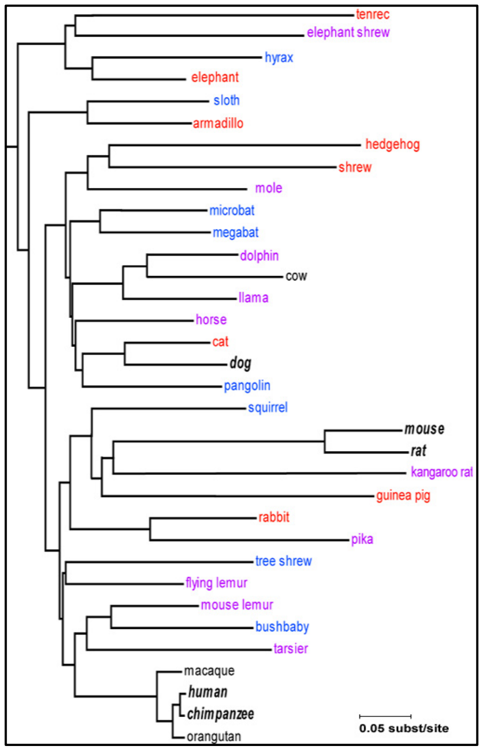
Getting started

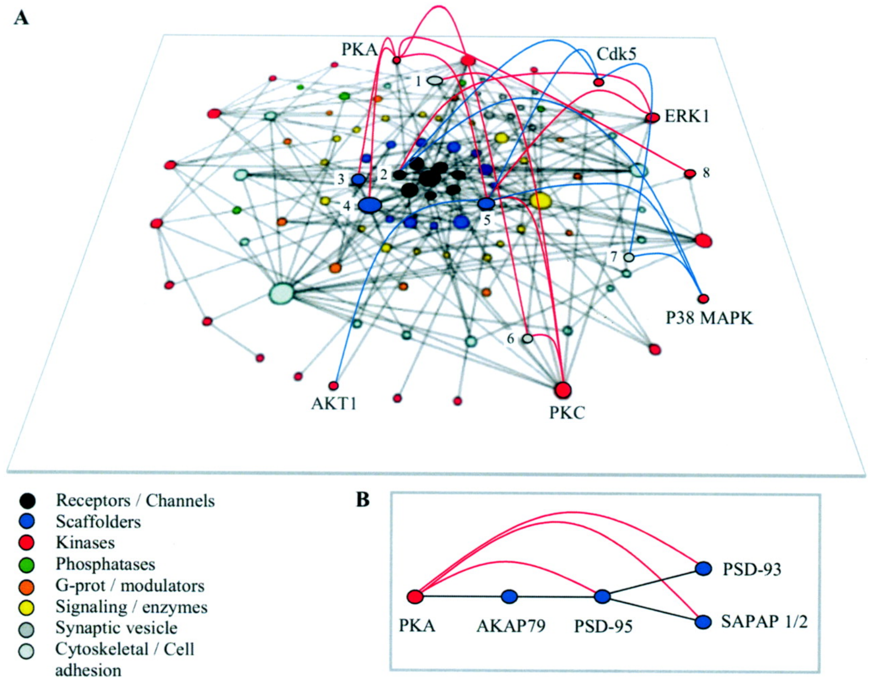
- Bioinformatics → solving biological problems by computers;

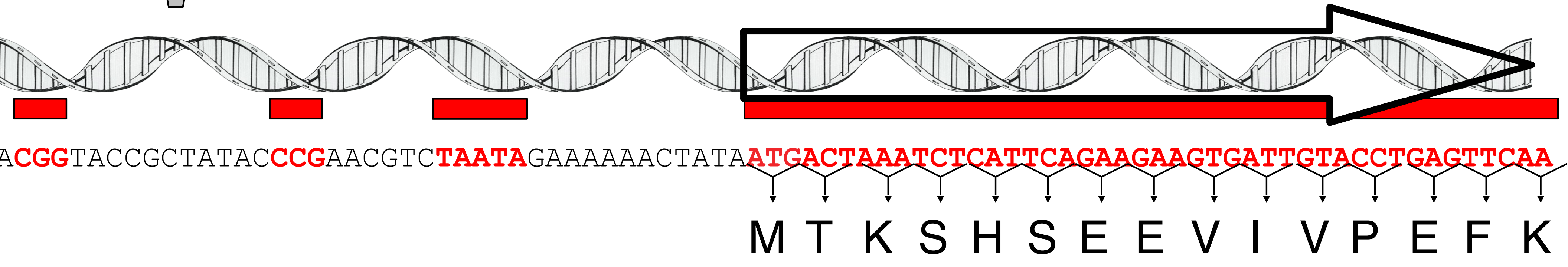
- Why computers?
  - Large amount of data
  - Complex computation

What kind of biological data?

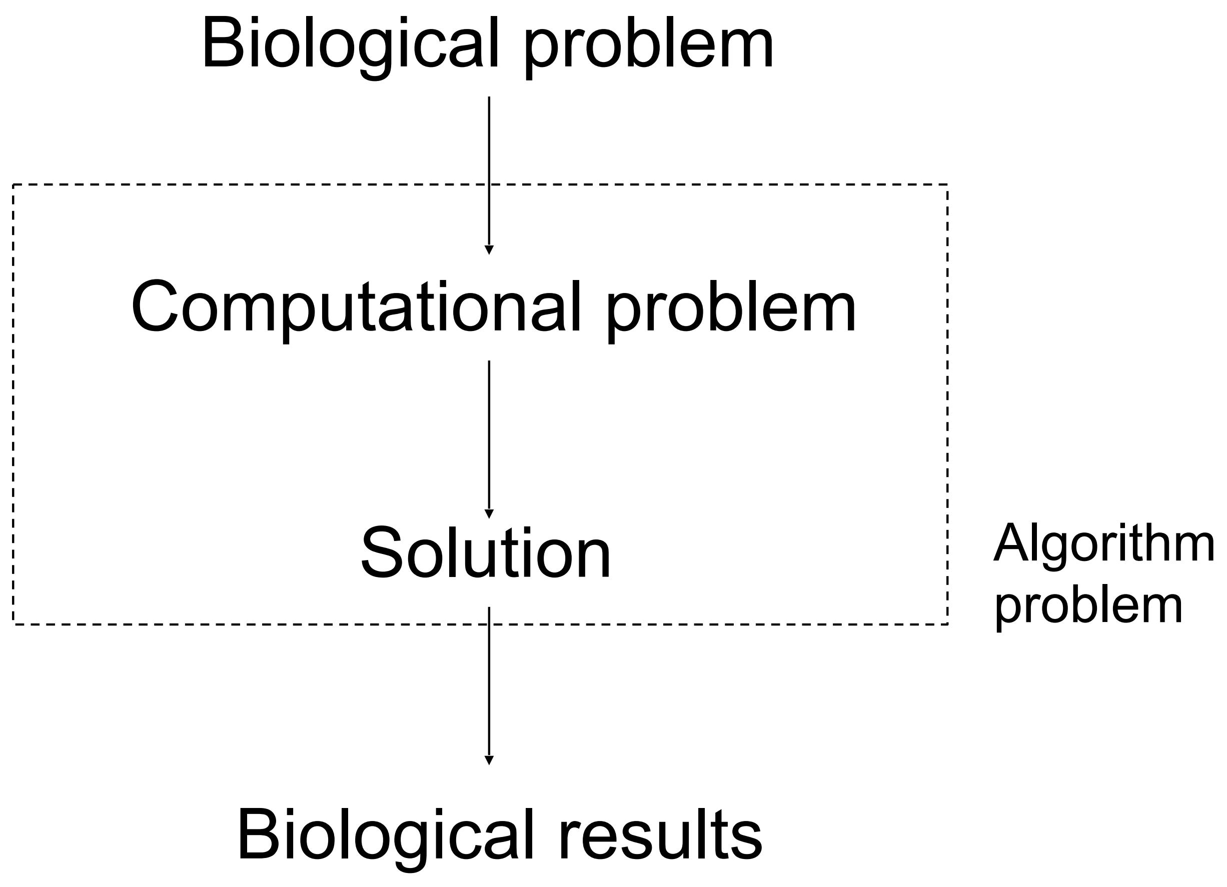




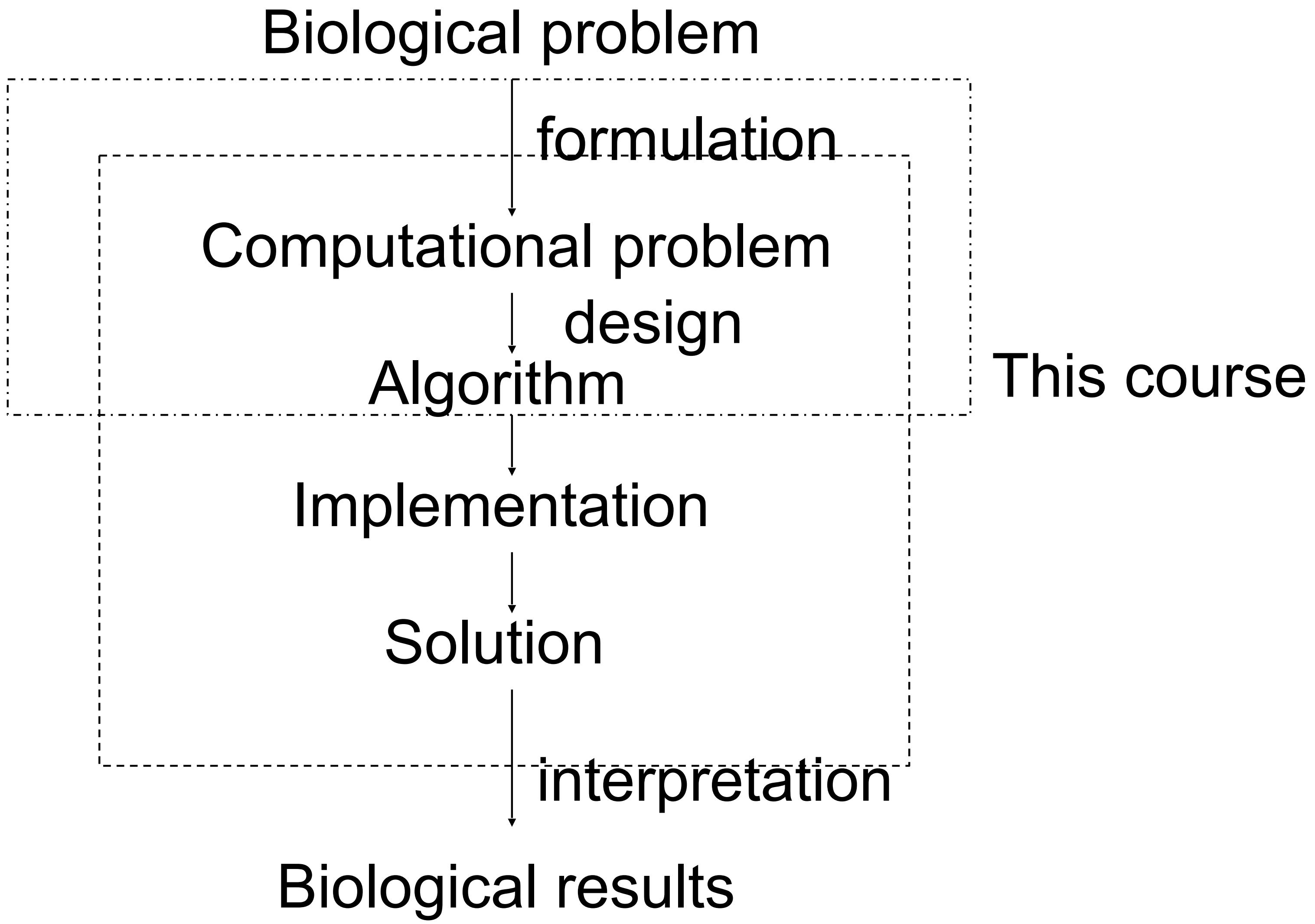



ACGGTACCGCTATACCCGAACGTCTAATAGAAAAAACTATAATGACTAAATCTCATTCAGAAGAAGTGATTGTACCTGAGTTCAA
M T K S H S E E V I V P E F K

A general solution pipeline for bioinformatics problems



A general solution pipeline for bioinformatics problems



What is a (computer) algorithm?

- An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required **output** for any legitimate **input** in a **finite** amount of time.

Features of Algorithm

- finiteness (otherwise: computational method)
  - termination
- definiteness
  - precise definition of each step
- input (zero or more)
- output (one or more)
- effectiveness

- Its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper. -- Knuth

Example of computational problem: sorting

- Statement of problem:
  - *Input:* A sequence of *n* numbers $<a_1, a_2, \ldots, a_n>$
  - *Output:* A reordering of the input sequence $<a'_1, a'_2, \ldots, a'_n>$ so that $a'_i \leq a'_j$, whenever $i < j$
- Instance:
  - Input <5, 3, 2, 8, 3>

Algorithms

- How to express algorithms
- How to design algorithms
- Proving correctness
- Efficiency
  - Theoretical analysis
  - Empirical analysis
- Optimality

How to express algorithms

- Idea
- Flowchart
- **Pseudocode**
- Program – implementation

How do we design an algorithm?

One step at a time – Iterative algorithms

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
codeC
<postCond>
```

Loop Invariants

Breaking problems down: recursive algorithms

Recursion: a subroutine which calls itself, with different parameters.

Compute $n! = \text{factorial}(n)$

$$\text{factorial}(n) = n\times (n-1) \times\ldots\times 2\times 1$$

$$= n\times\text{factorial}(n-1)$$

Suppose routine factorial(p) can find factorial of p for all $p < m$. Then factorial(m+1) can be solved as follows:

$$\text{factorial}(m+1) = (m+1) \times\text{factorial}(m)$$

```
Factorial(m)
{
    If $m = 1$, $\text{Factorial}(m) = 1$;
    else  $\text{Factorial}(m) = m \times \text{Factorial}(m-1)$
}
```

Rules:

Have a base case for which the subroutine need not call itself.
For general case, the subroutine does some operations, calls itself, gets result and does some operations with the result.

To get result, subroutine should progressively move towards the base case.

Recursion vs. iteration

```
Factorial_It(m)
{
prod = 1
For j=1  to m
   prod $\rightarrow$prod $*$j
}
```

In general, iteration is more efficient than recursion because of maintenance of state information.

Towers of Hanoi

Source peg, Destination peg, Auxiliary peg

k disks on the source peg. Need to move all k disks to the destination peg using the auxiliary peg, without ever keeping a bigger disk on the smaller disk.

Why can this be solved by recursion?

We know how to move 1 disk from source to destination.

For two disks, move the top one to the auxiliary, bottom one to the destination, then first to the destination.

For three disks, move top two disks from source to auxiliary, using destination.

Then move the bottom one from the source to the destination.

Finally move the two disks from auxiliary to destination using source.

We know how to solve this for k=1

Suppose we know how to solve this for k-1 disks.

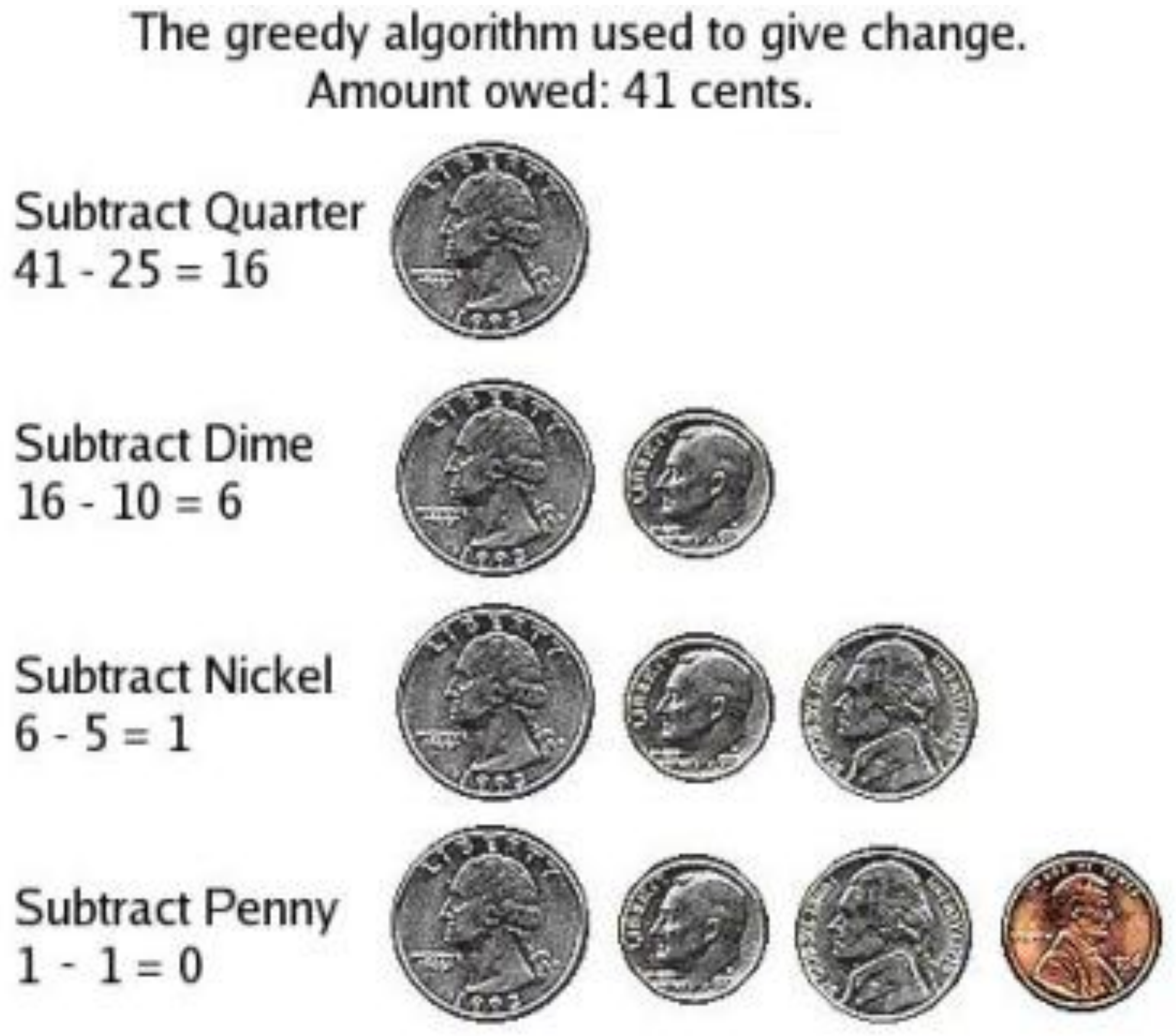
We will first move top k-1 disks from source to auxiliary, using destination.

Will move the bottom one from the source to the destination.

Will move the k-1 disks from auxiliary to destination using source.

```
TowerHanoi(k, source, auxiliary, destination)
  {
    If k=1 move disk from source to destination; (base case)
    else,
      {
          TowerHanoi(top k-1, source, destination, auxiliary);
          Move the kth disk from source to destination;
          Tower of Hanoi(k-1, auxiliary, source, destination);
      }
}
```

Get the biggest first – greedy approaches



The greedy algorithm used to give change.
Amount owed: 41 cents.

Subtract Quarter
41 - 25 = 16

Subtract Dime
16 - 10 = 6

Subtract Nickel
6 - 5 = 1

Subtract Penny
1 - 1 = 0

Coin change problem

Input: a number P (price)

Output: a way of paying P using bills ($5, $1) or coins (1, 5, 10, 25 cents)

Objective: minimize the number of coins used

8.37 = 5 + 1 + 1 + 1 + 0.25 + 0.10 + 0.01 + 0.01

8 bills/coins used

Coin change problem

Greedy algorithm: for simplicity we assume that the coin values are: 10,5,1 (cents)

```
GreedyCoinChange(N) {
  if N ≥ 10 then
    output(10), GreedyCoinChange(N-10)
  else if N ≥ 5 then
    output(5), GreedyCoinChange(N-5)
  else if N ≥ 1 then
    output(1), GreedyCoinChange(N-1)
}
```

Exact algorithm vs. heuristic algorithm

- Exact algorithm: guaranteed to report the correct (optimal) result
  - May be very expensive for some (intractable) problems
- Otherwise, heuristic algorithm
  - Many randomized algorithms
  - Judging by performance on real problems

Coin change problem

```
GreedyCoinChange(N) {
  if N ≥ 10 then
    output(10), GreedyCoinChange(N-10)
  else if N ≥ 5 then
    output(5), GreedyCoinChange(N-5)
  else if N ≥ 1 then
    output(1), GreedyCoinChange(N-1)
}
```

Theorem:
For coin values 10,5,1, the GreedyCoinChange algorithm outputs the minimal number of coins.

Coin change problem

Theorem: for coin values 10,5,1, the GreedyCoinChange algorithm outputs the minimal number of coins.

Proof:
Let N be the amount to be paid. Let the optimal solution be P=A*10 + B*5 + C. Clearly $B \leq 1$ (otherwise we can decrease B by 2 and increase A by 1, improving the solution). Similarly, $C \leq 4$.

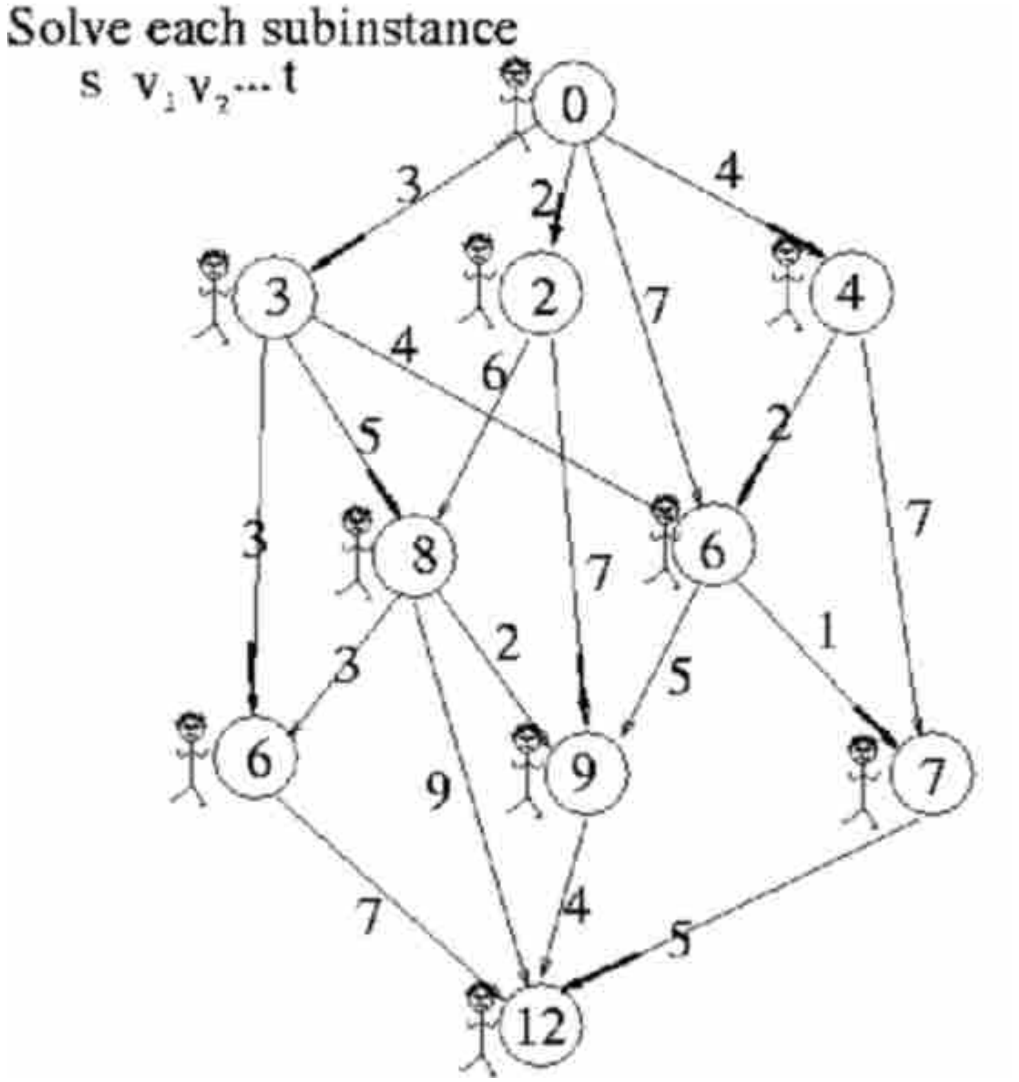
Let the solution given by GreedyCoinChange be P=a*10 + b*5 + c. Clearly $b \leq 1$ (otherwise the algorithm would output 10 instead of 5). Similarly $c \leq 4$.

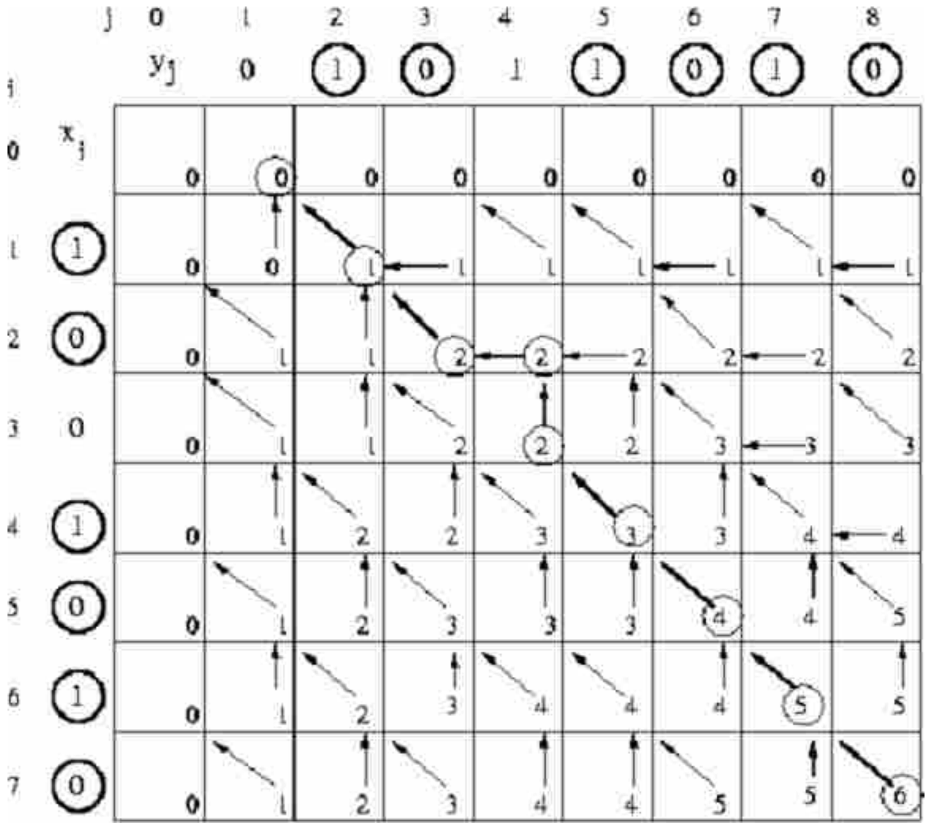
From $0 \leq C \leq 4$ and P=(2A+B)*5+C we have C=P mod 5.
Similarly c=P mod 5, and hence c=C. Let Q=(P-C)/5.

From $0 \leq B \leq 1$ and Q = 2A + B we have B=Q mod 2.
Similarly b=Q mod 2, and hence b=B.

Thus a=A, b=B, c=C, i.e., the solution given by GreedyCoinChange is optimal.

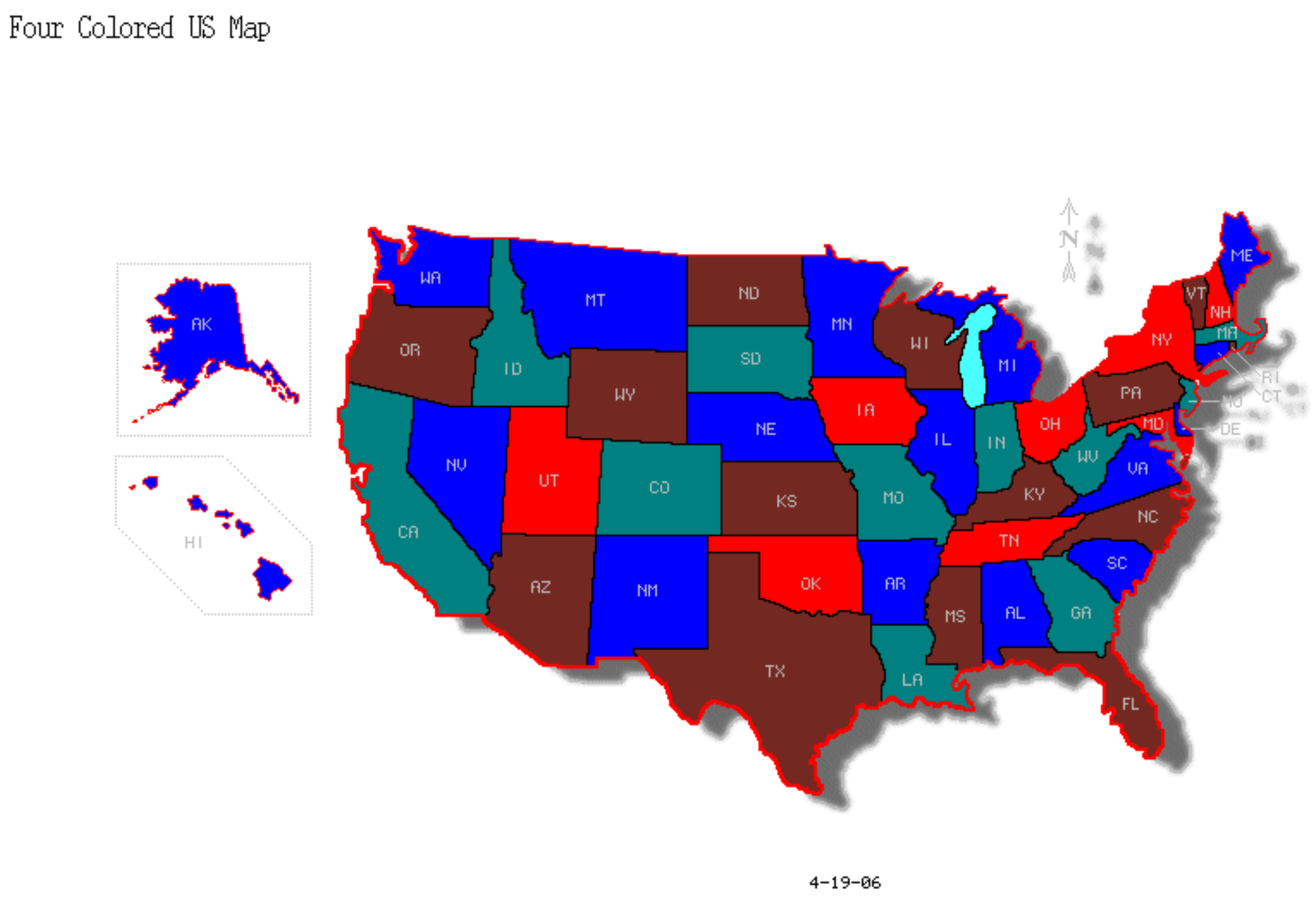
Breaking problems down: dynamic programming





	j	0	1	2	3	4	5	6	7	8
i		y_j	0	1	0	1	1	0	1	0
0	x_i	0	0	0	0	0	0	0	0	0
1	1	0	0	1	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	2	2
3	0	0	1	1	2	2	2	3	3	3
4	1	0	1	2	2	3	3	3	4	4
5	0	0	1	2	3	3	3	4	4	5
6	1	0	1	2	3	4	4	4	5	5
7	0	0	1	2	3	4	4	5	5	6

Abstract thinking: graph algorithms



Analysis of Algorithms

- How good is the algorithm?
  - Correctness
  - Time efficiency
  - Space efficiency

- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

Correctness

- Termination
  - Well-founded sets: find a non-negative quantity that always decreases as the algorithm is executed
- Partial Correctness
  - For recursive algorithms: induction
  - For iterative algorithms: axiomatic semantics, loop invariants

Complexity

- Space complexity

- Time complexity
  - For iterative algorithms: sums
  - For recursive algorithms: recurrence relations

Shortest path problem

A city has n view points

Buses go from one view point to another

A bus driver wishes to follow the shortest path
(travel time wise).

Every view point is connected to another by a road.

However, some roads are less congested than others.

Also, roads are one-way, i.e., the road from view point 1 to
2, is different from that from view point 2 to 1.

How to find the shortest path between any two pairs?

Naïve approach:

List all the paths between a given pair of view points

Compute the travel time for each.

Choose the shortest one.

How many paths are there between any two view points?

$$n! \cong (n/e)^n$$

Will be impossible to run your algorithm for n = 30.

Machine independent analysis

We assume that every basic operation takes constant time:

Example Basic Operations:

Addition, Subtraction, Multiplication, Memory Access

Non-basic Operations:

Sorting, Searching

Efficiency of an algorithm is measured by the number of basic operations it performs

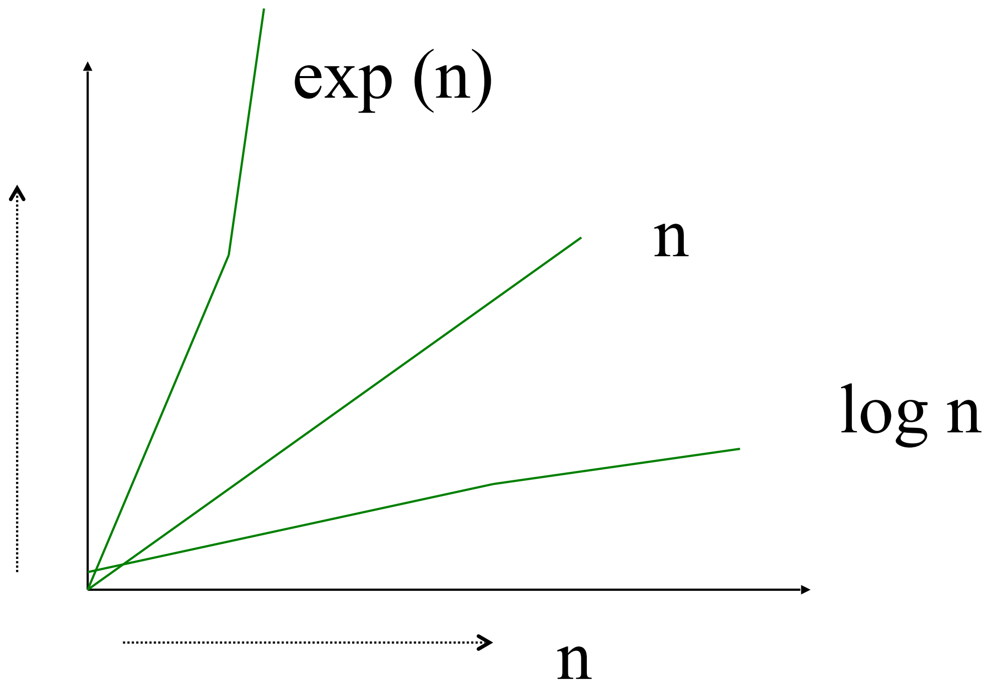
We do not distinguish between the basic operations.

Order of Increase

We care about the speed of our algorithms for large input sizes.

Note that for large n, log(n)/n , and n/exp(n) are very small.

However, n/2n is a constant for all n.



Function Orders

A function f(n) is O(g(n)) if "increase" of f(n) is not faster than that of g(n).

A function f(n) is O(g(n)) if there exists a number n_0 and a nonnegative constant *c* such that for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$.

If $\lim_{n \to \propto} f(n)/g(n)$ exists and is finite, then f(n) is O(g(n))

Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , n + n^2

$\lim_{n \rightarrow \propto}$ sqrt(n) /n = 0,	sqrt(n) is O(n)
$\lim_{n \rightarrow \propto}$ n/sqrt(n) = infinity,	n is not O(sqrt(n))
$\lim_{n \rightarrow \propto}$ n /2n = 1/2,	n is O(2n)
$\lim_{n \rightarrow \propto}$ 2n /n = 2,	2n is O(n)

$\lim_{n \to \propto} \ln(n)/n = 0$,	$\ln(n)$ is $O(n)$
$\lim_{n \to \propto} n/\ln(n) = \text{infinity}$,	n is not $O(\ln(n))$
$\lim_{n \to \propto} \exp(n)/n = \text{infinity}$,	$\exp(n)$ is not $O(n)$
$\lim_{n \to \propto} n/\exp(n) = 0$,	n is $O(\exp(n))$
$\lim_{n \to \propto} (n+\text{sqrt}(n))/n = 1$,	$n + \text{sqrt}(n)$ is $O(n)$
$\lim_{n \to \propto} n/(\text{sqrt}(n)+n) = 1$,	n is $O(n+\text{sqrt}(n))$
$\lim_{n \to \propto} (n + n^2)/n = \text{infinity}$,	$n + n^2$ is not $O(n)$
$\lim_{n \to \propto} n/(n + n^2) = 0$,	n is $O(n + n^2)$

Implication of O notation

Suppose we know that our algorithm uses at most O(f(n)) basic steps for any n inputs, and n is sufficiently large, then we know that our algorithm will terminate after executing at most constant times f(n) basic steps.

We know that a basic step takes a constant time in a machine.

Hence, our algorithm will terminate in a constant times f(n) units of time, for all large n.

Other Complexity Notation

Intuitively, (not exactly) f(n) is O(g(n)) means f(n) ≤ g(n) →g(n) is an upper bound for f(n).

Now a lower bound notation, Ω(n)

f(n) is Ω(g(n)), if f(n) ≥*c*g(n) for some positive constant *c*, and all large n.

$\lim_{n\to\propto} (f(n)/g(n)) > 0$, if $\lim_{n\to\propto} (f(n)/g(n))$ exists f(n) is $\theta(g(n))$ if f(n) is $O(g(n))$ and $\Omega(g(n))$

$\theta(g(n))$ is ``asymptotic equality''

$\lim_{n \to \propto} (f(n)/g(n))$ is a finite, positive constant, if it exists

f(n) is $o(g(n))$ if f(n) is $O(g(n))$ but not $\Omega(g(n))$

``asymptotic strict inequality''

f(n) is $o(g(n))$ if given any positive constant c, there exists some m such that $f(n) < cg(n)$ for all $n \geq m$

$\lim_{n \to \propto} (f(n)/g(n)) = 0$, if $\lim_{n \to \propto} (f(n)/g(n))$ exists

Asymptotically less than or equal to	O
Asymptotically greater than or equal to	Ω
Asymptotically equal to	θ
Asymptotically strictly less	o

Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , n + n^2

$\lim_{n\to\propto}$ sqrt(n) /n = 0, sqrt(n) is o(n)

$\lim_{n\to\propto}$ n/sqrt(n) = infinity, n is Ω(sqrt(n))

$\lim_{n\to\propto}$ n /2n = 1/2, n is θ(2n), Ω(2n)

$\lim_{n\to\propto}$ 2n /n = 2, 2n is θ(n), Ω(n)

Implication of the O, Ω Notation

Suppose, an algorithm has complexity O(f(n)) . This means that there exists a positive constant *c* such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at most *c*f(n) steps.

Suppose, an algorithm has complexity Ω(f(n)) . This means that there exists a positive constant *c* such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at least *c*f(n) steps.

Complexity of a problem vs. complexity of an algorithm

A problem is $O(f(n))$ means there is some $O(f(n))$ algorithm to solve the problem.

A problem is $\Omega(f(n))$ means every algorithm that can solve the problem is $\Omega(f(n))$

Worst case vs. average case

Suppose, an algorithm has complexity O(f(n)) . This means that there exists a positive constant *c* such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at most *c*f(n) steps.

Worse case complexity!

Much harder to analyze the average case complexity of an algorithm!

Algorithm: conclusions

- Finiteness
  - terminates after a finite number of steps
- Definiteness
  - rigorously and unambiguously specified
- Input
  - valid inputs are clearly specified
- Output
  - can be proved to produce the correct output given a valid input
- Effectiveness
  - steps are sufficiently simple and basic