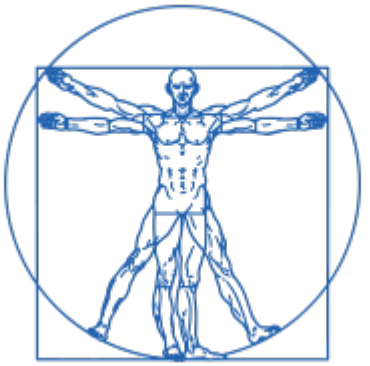
Today's objectives

- What is computational biology used for with historical context
- Types of algorithms and Big-O notation
- String search

Office hours (zoom link on blackboard announcements):
Fridays 1:00-2:00pm (Emiliano)
Mondays 3:30-4:30pm (Faye)

Biology
(and life on mars)

TATCGAATCGTGTGA
TACCGCGATAAACCT
AATTTTGAGGTTCGC
CATCCTCTGGCTATT
CCGCCTCAGTTTGA



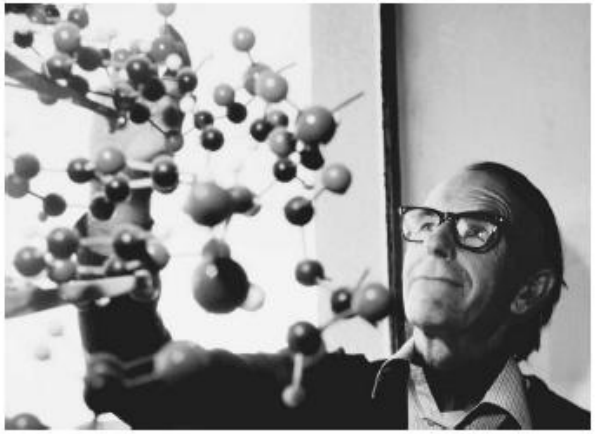
Genome sequencing and assembly

Genome annotation & function

Genome variation (evolution + disease)

The birth of computational biology

- Fred Sanger sequenced the first protein, insulin, in 1952 for which he received the Nobel prize. The Cytochrome C protein was sequenced from multiple organisms (Margoliash et al. 1961).
- Margaret Dayhoff & colleagues stored protein sequence data in the Protein Information Resource (1960s & 1970s)
- Dayhoff & others organized sequences into families & subfamilies, and derived a phylogenetic tree of proteins.

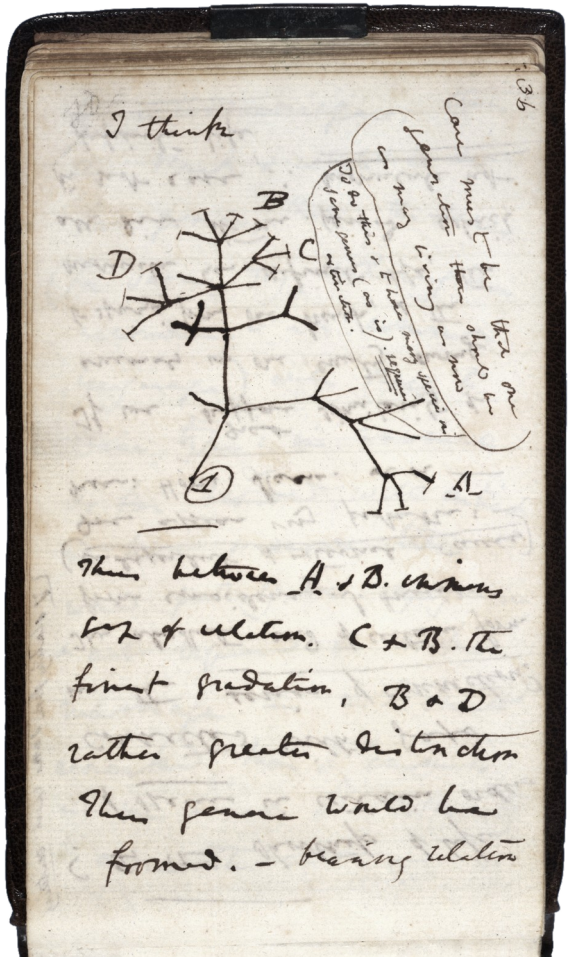


Fred Sanger

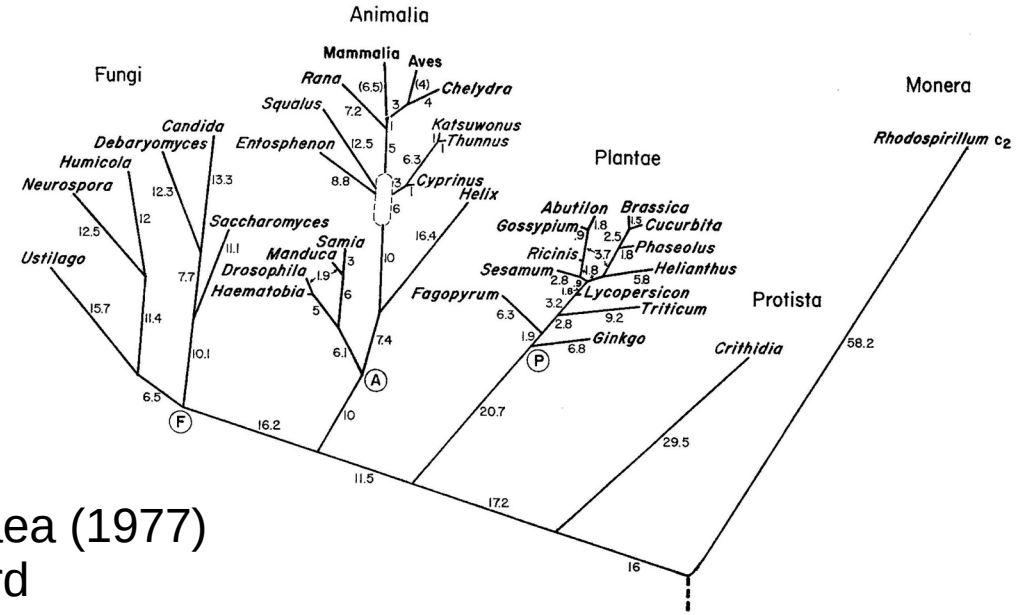
Margaret Dayhoff

First use of computers for phylogenetics

Page from Darwin's notebooks around July 1837 showing his first sketch of an evolutionary tree



Eukaryotic evolutionary tree by Dayhoff & colleagues (1973)



Archaea (1977)
Asgard

Fig. 2. The detailed cytochrome *c* evolutionary tree. The order of branching for the five kingdoms is the same as configuration 1 in Fig. 4. The progression of time is toward the top of the tree. The lengths of the branches are drawn in proportion to the numbers beside the branches, which are PAMs or Accepted Point Mutations estimated to have occurred on these branches

Implications: molecular phylogenetics

Sequencing time-line

1977 -- Frederick Sanger develops a DNA sequencing technique uses it to sequence the first full genome – that of a virus called phiX174 (3,569 bp). Nobel prize

1983 -- The polymerase chain reaction (PCR) is developed – a technique used for amplifying DNA – by Dr Kary Mullis at the Cetus Corporation in California, USA. Nobel prize

1986 -- Automated sequencing machine (Lee Hood).

1995 -- The first bacterium genome sequence is completed (Haemophilus influenza), 1.8 Mbp.

1996 -- Genome of yeast, Saccharomyces cerevisiae, 12.5 Mbp.

1998 -- C. elegans, 100 Mbp

2000 -- A. thaliana, 119 Mbp

2000 -- Drosophila melanogaster (fruit fly) is completed, 165 Mbp

2002 -- Mouse genome, 3.5 Gbp

2003 -- Human genome, 3.2 Gbp

2004 -- ENCODE project

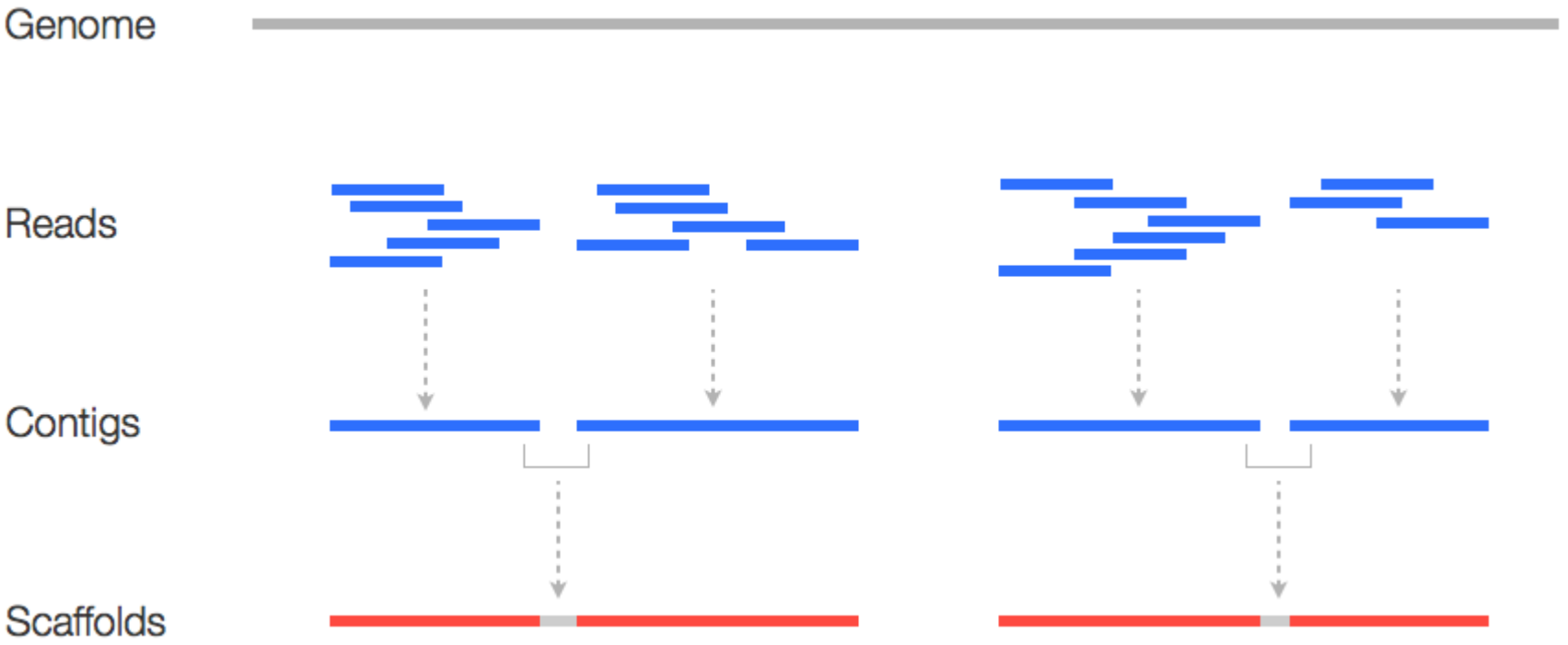
2005 -- HapMap project

2007 -- Next Generation sequencing

2008 -- 1000 human genomes project launched

2009 -- Cancer genome project

Genome assembly



Alignment

Needleman–Wunsch global alignment

Smith-Waterman local alignment

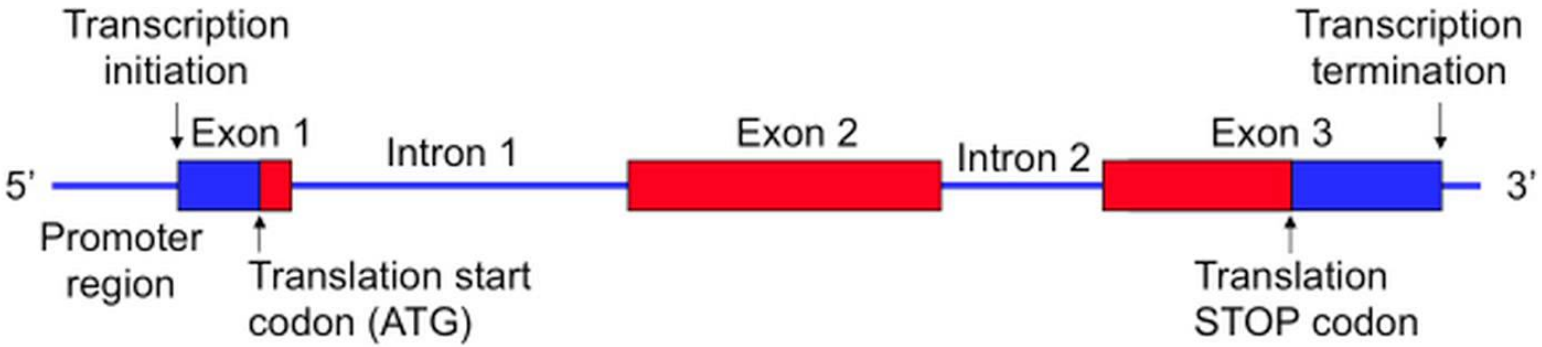
Assembly

Overlap-Layout-Consensus assembly

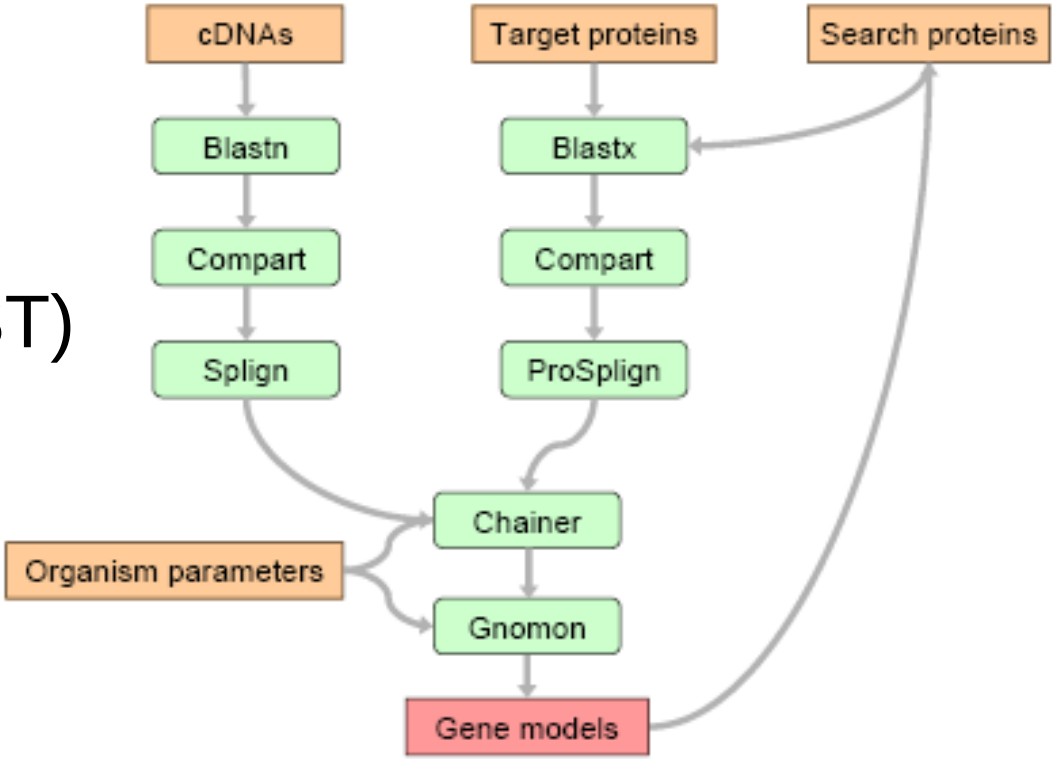
De Bruijn graph - directed graph representing overlaps between sequences of symbols

The next challenge:
Polyploid assembly

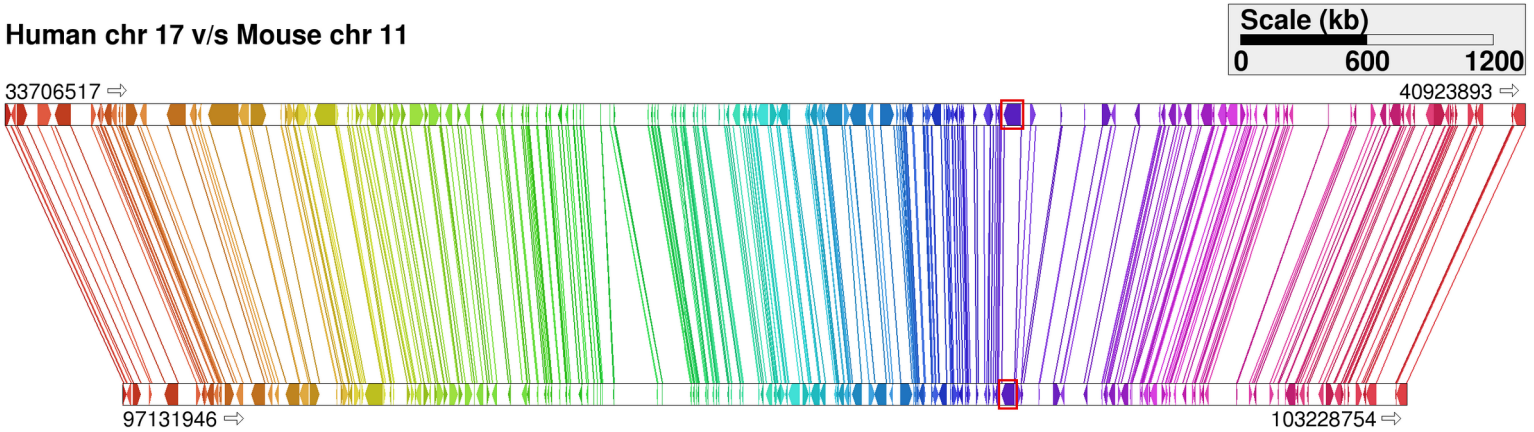
Gene prediction

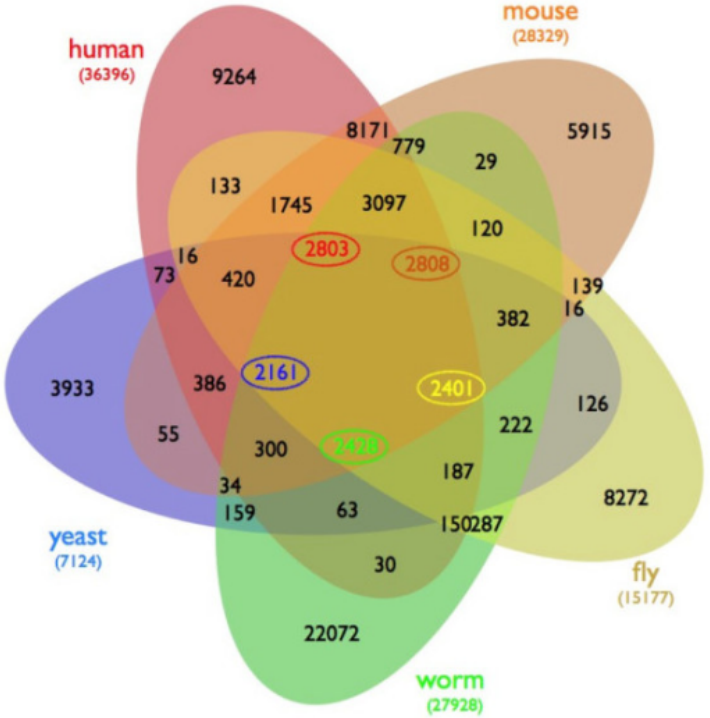


Predictions
- ab initio (HMM)
- sequence similarity (BLAST)
- cDNA (Splign)



Gene annotation and function via model organisms

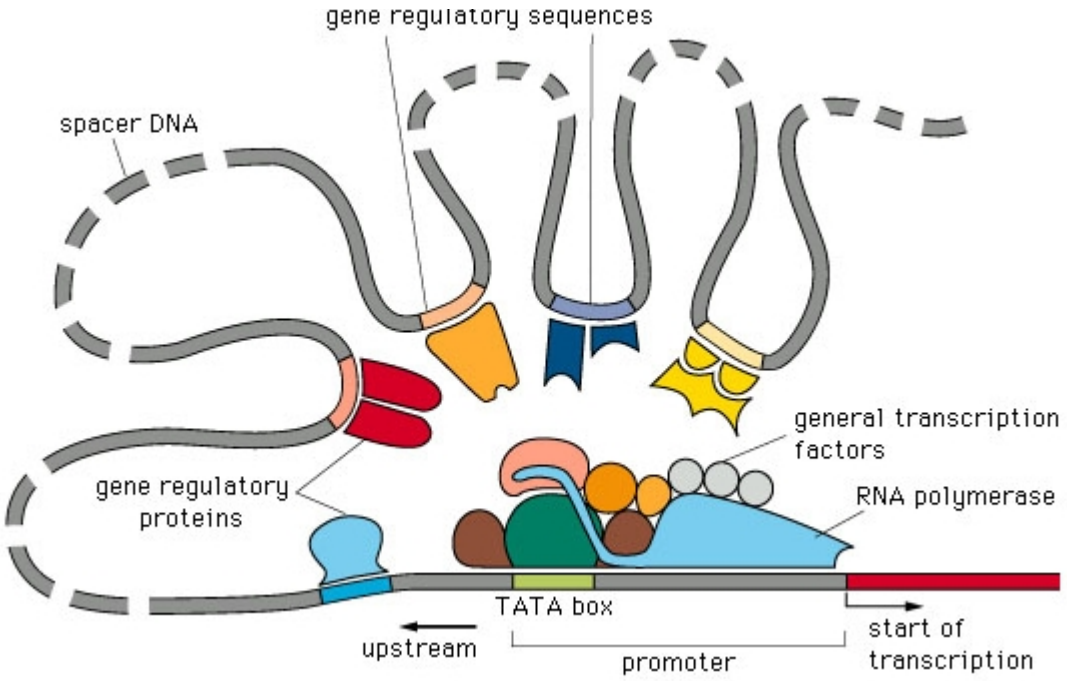


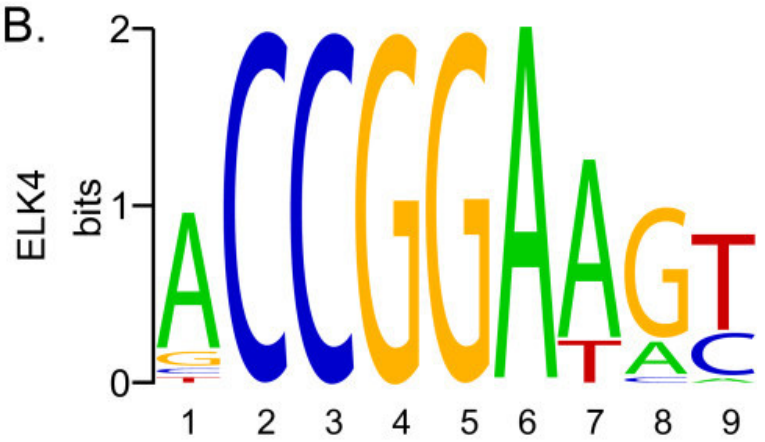


Alignment
BLAST search algorithm

Protein domains
IntroPro database

Regulatory sequences

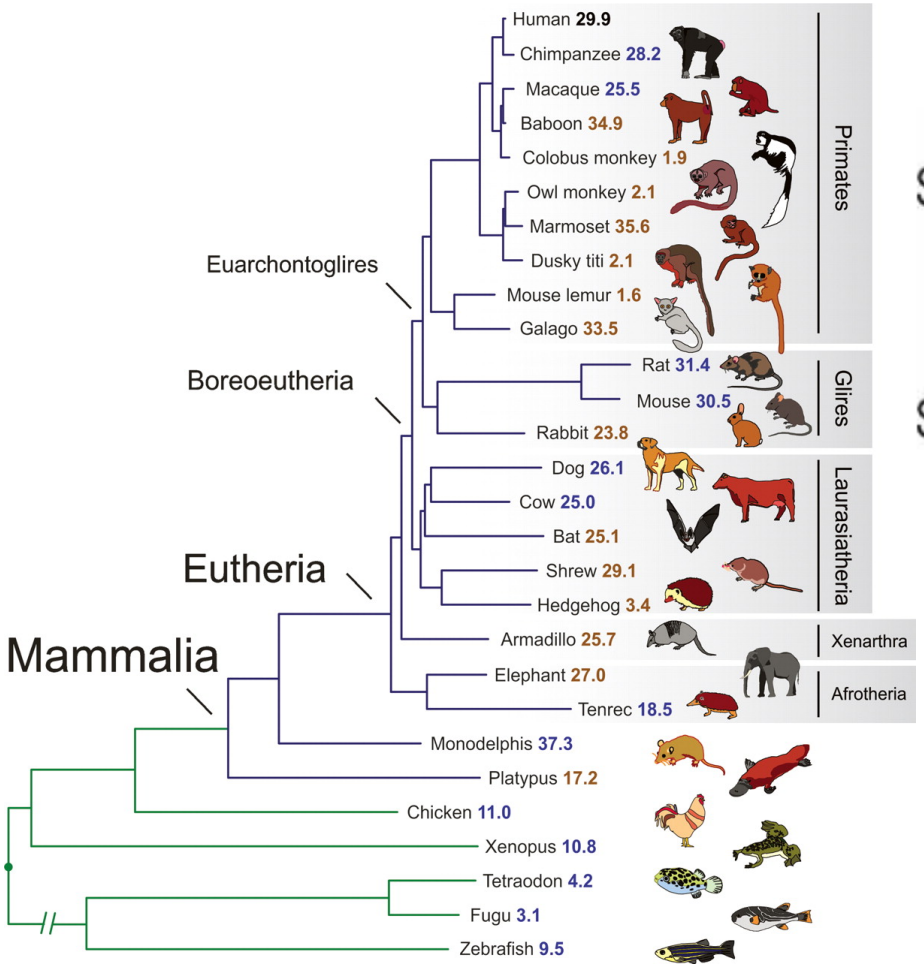


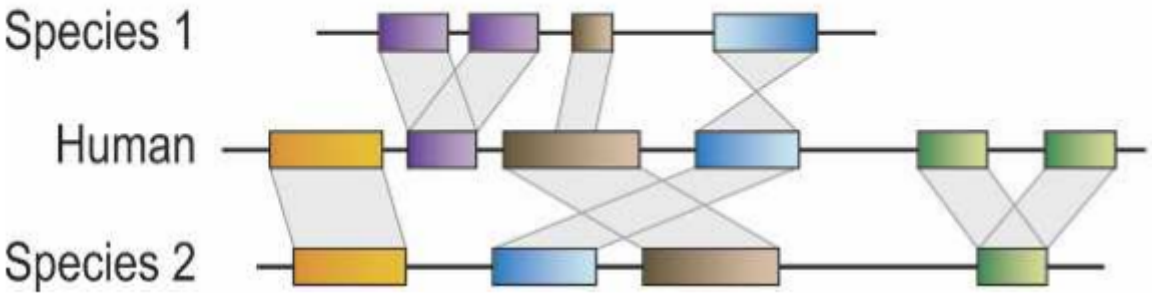


Motif models
EM algorithm
Gibbs sampling
Machine learning

Binding site (motif) conservation across species

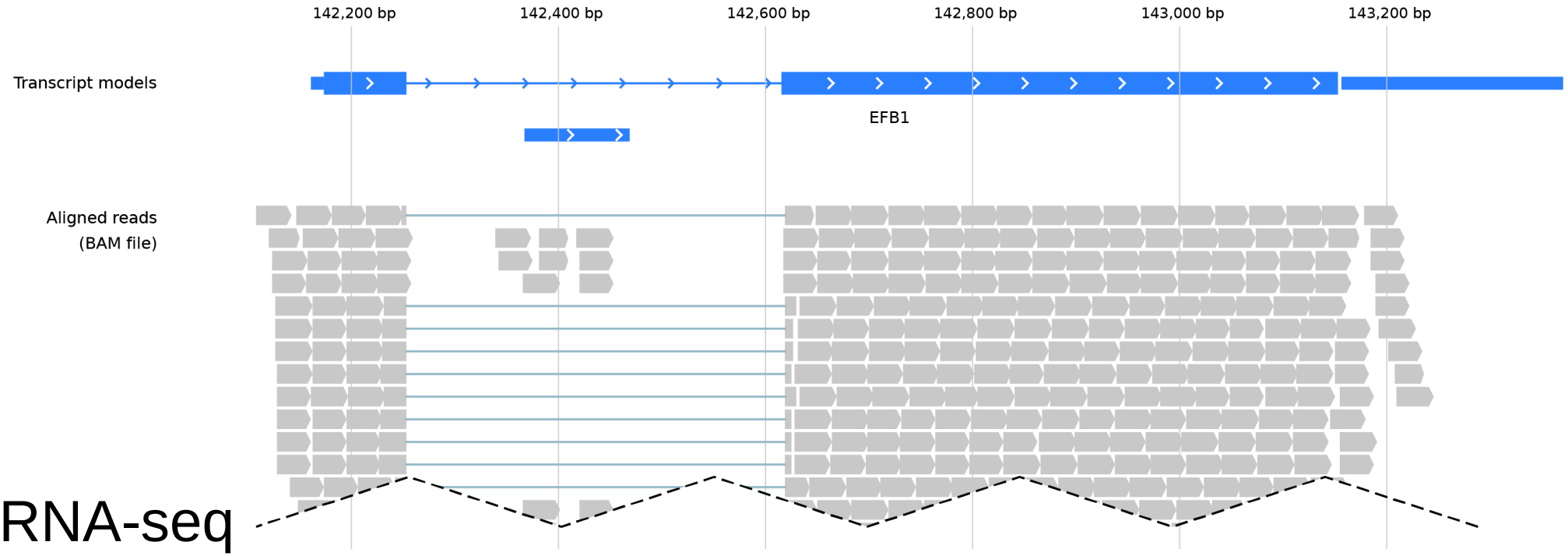
Comparative genomics





Phylogenetic methods
Tree algorithms
Maximum likelihood
Markov chain Monte Carlo

Functional annotation



RNA-seq
ChIP-seq
Bisulfite-seq
Ribo-seq
ATAC-seq

Short read mapping algorithms
Burrows-Wheeler Transform
HMMs

Genome variation

- Single nucleotide polymorphisms (SNPs)
- Insertion/deletion polymorphisms (InDels)
- Copy number variants (CNVs)
- Structural variation (e.g. translocation, inversion, large InDels and CNVs)

Bayesian methods
Likelihood models
Phylogenetic models

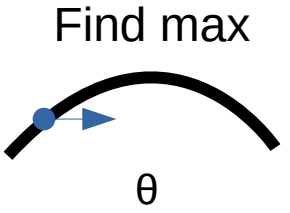
Reference: CGATGCATGCATATGA
Alt Allele: C--TGCATTCATATGA

Today's objectives

- What is computational biology used for with historical context
- Types of algorithms and Big-O notation
- String search

Types of algorithms

- Find (number or positions) of a string (e.g. AGTG) in a sequence. (exhaustive)
- Find parameter (θ) that optimizes a function P(protein binding) = f(θ). (greedy)



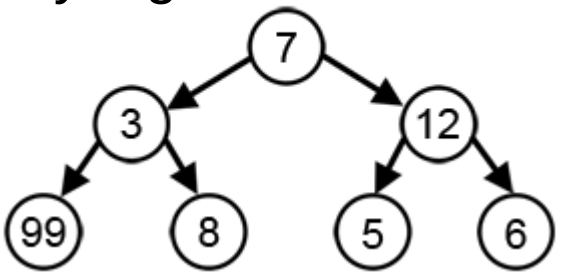
- Sort a list of numbers. (recursion)
- Find the best tree given an alignment (branch and bound)

- Different types of algorithms can be used to solve the same problem
- Algorithms differ in time/memory but also their solution

Types of algorithms

- exhaustive search (brute force) – systematically enumerating all possible candidates for the solution and checking (minimum path)
- branch and bound – solutions form a tree relationship, before enumerating a branch of solutions upper and lower bounds are first checked to see if a possible solution exists (minimum path)
- dynamic programming – breaking problem into sub-problems, storing solutions from prior sub-problems (sum of each path)
- greedy algorithm – problem solving heuristic of making a local optimal choice at each stage with the hope of finding a global optimum (maximum path)
- randomization – use a random number to make a decision, randomness either affects run time or output (which path sum = 18)
- heuristic – a decision or solution that may not be optimal but is fast or close enough to optimal (practical), may rely on intuition or analogy rather than proof (greedy algorithm, minimum path)

binary tree is a tree data structure in which each node has at most two children



Types of algorithms

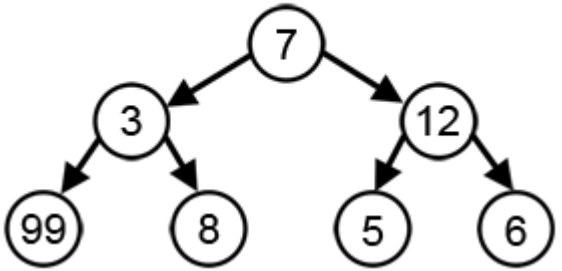
- exhaustive search (brute force) – systematically enumerating all possible candidates for the solution and checking (minimum path)

7 + 3 + 99 = 109
7 + 3 + 8 = 18
7 + 12 + 5 = 24
7 + 12 + 6 = 25

Guaranteed best solution, but can be limited by time or memory.

Sort (9, 2, 5, 1), exhaustive search doesn't make sense

binary tree is a tree data structure in which each node has at most two children



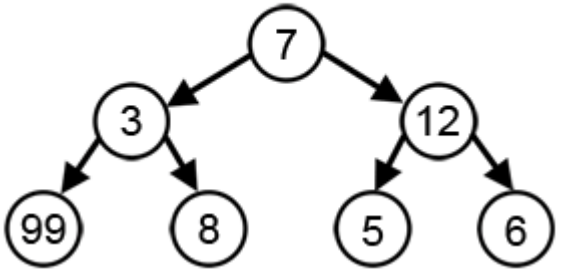
Types of algorithms

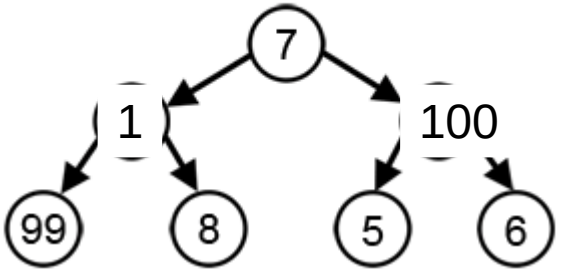
- branch and bound – solutions form a tree relationship, before enumerating a branch of solutions upper and lower bounds are first checked to see if a possible solution exists (minimum path)

Suppose nodes can have values 1..100
Node 7: min = 7 + 1 + 1, max = 7 + 100 + 100
Node 3: min = 7 + 3 + 1, max = 7 + 3 + 100 [11,110]
Node 12: min = 7 + 12 + 1, max = 7 + 12 + 100 [20,119]

What if node 12 = 100 and node 3 = 1?
Node 1: min = 7 + 1 + 1, max = 7 + 1 + 100 [9,108]
Node 100: min = 7 + 100 + 1, max = 7 + 100 + 100 [108,207]

binary tree is a tree data structure in which each node has at most two children





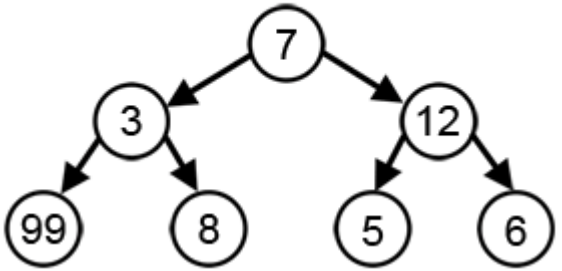
Types of algorithms

- dynamic programming – breaking problem into sub-problems, storing solutions from prior sub-problems (find sum of each path)

Exhaustive:
Path 1: 7 + 3 + 99
Path 2: 7 + 3 + 8
Path 3: 7 + 12 + 5
Path 4: 7 + 12 + 6

Dynamic: add left and right nodes
7 + 3 = 10; 10 + 99 (P1), 10 + 8 (P2)
7 + 12 = 19; 19 + 5 (P3), 19 + 6 (P4)

binary tree is a tree data structure in which each node has at most two children

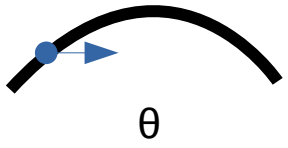


Types of algorithms

- greedy algorithm – problem solving heuristic of making a local optimal choice at each stage with the hope of finding a global optimum (maximum path)

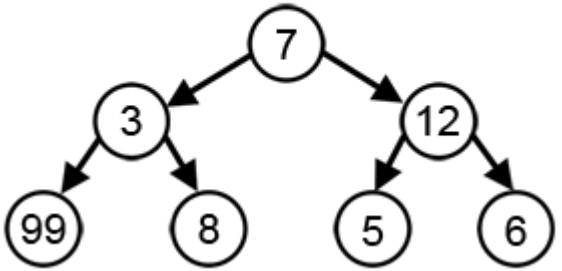
Greedy (pick larger descendant)

7 + 12 + 6 = 25 (incorrect maximum path)



Example: maximize f(θ)

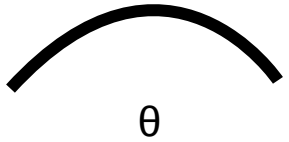
binary tree is a tree data structure in which each node has at most two children



Types of algorithms

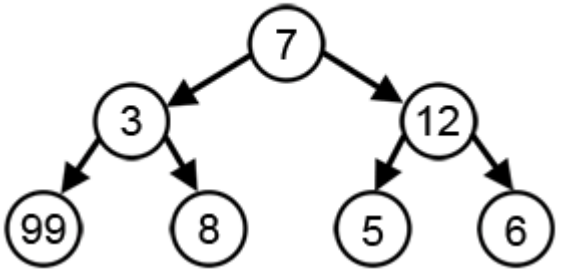
- randomization – use a random number to make a decision, randomness either affects run time or output (which path sum = 18)

  Can be used to find solutions to otherwise intractable problems (MCMC)



- heuristic – a decision or solution that may not be optimal but is fast or close enough to optimal, may rely on intuition or analogy rather than proof (greedy algorithm, minimum path)

binary tree is a tree data structure in which each node has at most two children



Types of algorithms

Exact vs approximate

- find best alignment of a sequence to the genome (BLAST=approximate)
- can be deterministic or not

Deterministic vs stochastic (non-deterministic)

- deterministic: same decision at every step, always the same output given an input (BLAST)
- stochastic: depends on some random (but informed) choices, can have different results from different runs

Today's objectives

- What is computational biology used for with historical context
- Types of algorithms and Big-O notation
- String search

Big O notation to quantify complexity (time or memory)

- Computers differ in speed, hence a comparable measure of an algorithm's speed is the number of operations that an algorithm makes not the time needed to make them.

- Algorithm's running time and memory usage is often related to size of the input, e.g. sort n integers.

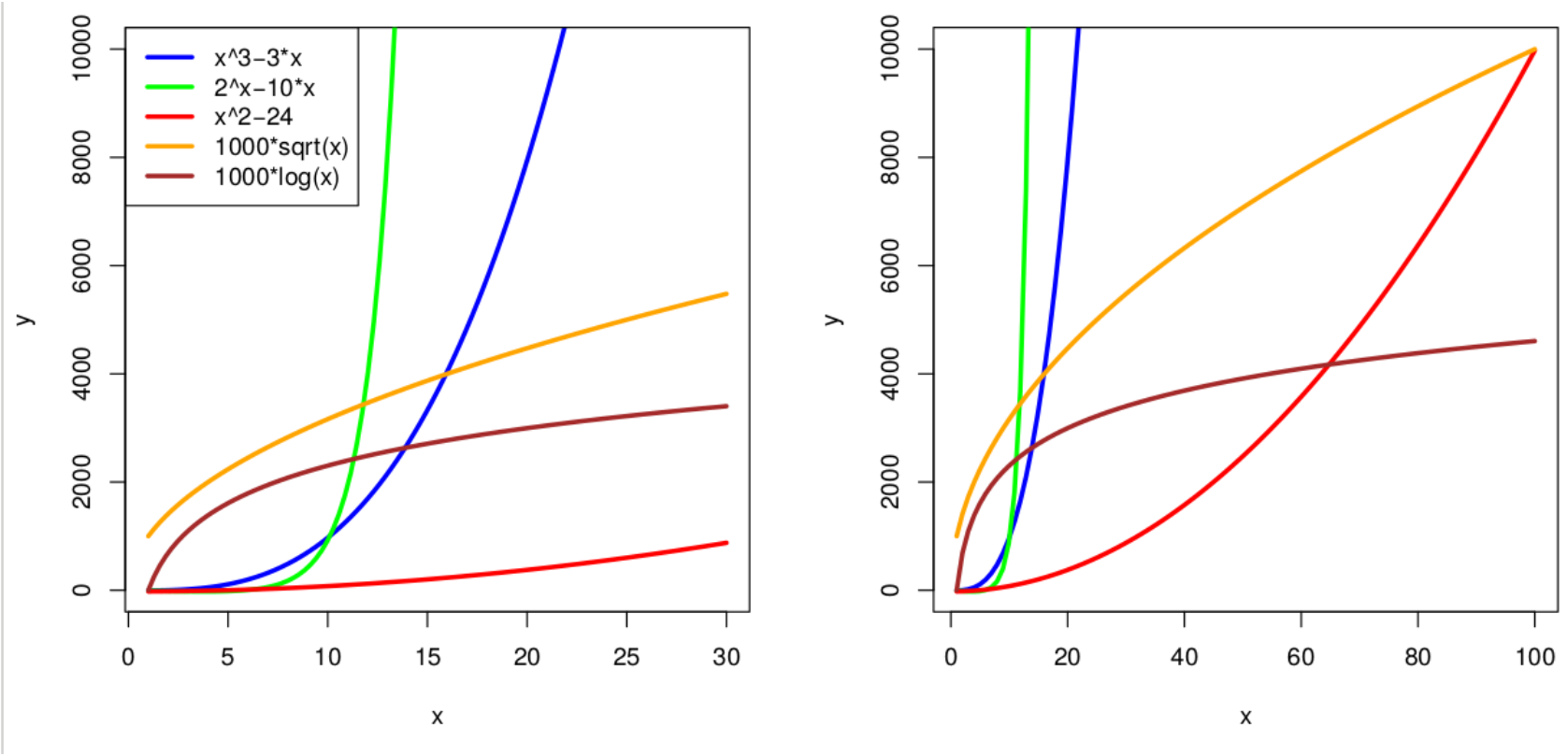
- Big O notation is a method of concisely defining the running time or memory usage of an algorithm (there is often a trade-off)

- Big O notation characterizes functions according to how they grow with input/parameter. The letter O is used because the growth rate of a function is also referred to as the order of approximation of the function, and typically describes an upper bound on the growth rate.

Big O notation examples

- If running time is quadratic, $O(n^2)$, it means that running an algorithm of input size n is limited by some quadratic function of n.
- For example: $99.7n^2$ or $0.001n^2$ or $5n^2+3.2n+99993$
- Big O notation $f(n) \sim O(n^2)$ tells us that the function f(n) does not grow faster than some quadratic function, in the worst case scenario.
- The function is typically chosen to be as simple as possible, omitting constant factors and lower order terms.
- $5n^2+3.2n+99993$
  - When n is small, e.g. 2, the third term (99993) dominates
  - When n is large, e.g. 1000, the first term (n^2) dominates

Run time

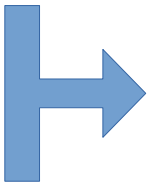


(best) O(1) = constant
O(log n) = logarithmic
$O(n^2)$ = quadratic
$O(n^x)$ = polynomial
(worst) $O(c^n)$ = exponential

Python functions

```
[2]: myarray = (1, 2, 8, 10)

[3]: print(myarray)
(1, 2, 8, 10)

[4]: print(myarray[0])
1

[9]: print(myarray[0:3])
(1, 2, 8)

[13]: for i in range(1,5):
          print(i)
1
2
3
4

[12]: for i in myarray:
          print(i)
1
2
8
10
```

```
[15]: j = 5
print(j)
for i in myarray:
    j = j+i
    print(j)
j = 50
print(j)
5
6
8
16
26
50
```



for loop

How many operations?
4 outside loop
2 within loop
O(length(myarray))
How much memory?
integer (j)
myarray (n)
O(length(myarray))

Big O notation

```
x = 1
x = 2
x = 3
y = 1
```

Time: 4 lines
Mem: 2 variables

```
for i = 1 to n
    x = 1
```

Time: 2 lines * n loops
Mem: 2 variables (i, x)

```
for i = 1 to n
    x[i] = 1
```

Time: 2 lines * n loops
Mem: 1 variable (i) one array (x) of length n

```
for i = 1 to n
    for j = 1 to n
        x[i,j] = 1
```

Time: 3 lines * n*n loops
Mem: 2 variables (i,j) one matirx (x) of size n*n

Time complexity

- Count operations (lines of code)
- For code within a loop, multiple lines by number of loops
- Loops within loops are multiplied
- Sequential loops are added

Memory complexity

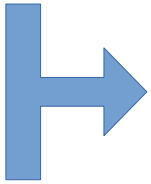
- Determine array or matrix size
- Examine assignments (=)
- Length of array or Row * Columns of matrix

Loops and operations

```
[2]: myarray = (1, 2, 8, 10)

[3]: print(myarray)

(1, 2, 8, 10)
```

```
[15]: j = 5
print(j)
for i in myarray:
    j = j+i
    print(j)
j = 50
print(j)
```

```
5
6
8
16
26
50
```



for loop

Time complexity

- Count operations (lines of code)
- For code within a loop, multiple lines by number of loops
- Loops within loops are multiplied
- Sequential loops are added

How many operations?

4 outside loop

2 (or 3) within loop * length(myarray)

= O(4+2*length(myarray)

= O(length(myarray))

Memory complexity

- Determine array or matrix size
- Examine assignments (=)
- Length of array or Row * Columns of matrix

How much memory?

integer (j) assignment

length(myarray)

= O(1 integer + length(myarray))

= O(length(myarray))

Time complexity

```
FunctionFactors(n):
    For i = 1 to n
        x = n*n
        print ‘x’
```

2 lines of code * n loops
= O(2n)
= O(n)

```
FunctionCombination2(n):
    For i = 1 to n-1
        j = 0
        While j < n
            j = j + 1
            print ‘i, j’
```

(1 line of code+while loop) * (n-1) loops
while loop = 2 lines * n loops (j=0,1,2,3... n-1)
= O((1+ 2*n)*(n-1))
= O(n - 1 + $2n^2$ + 2)
= O(n^2)

```
FunctionCombination3(n, m):
    For i = 1 to n
        For j = 1 to m
            print ‘i, j’
        For k = 1 to m
            print ‘i, k’
```

(j loop + k loop) * (n) loops
j loop = 1 line * m loops (j=1,2,3... m)
k loop= 1 line * m loops
= O((m loops + m loops)*(n))
= O(2m*n)
= O(m*n)

Memory complexity

```
StringSearch(n, m):
    For i = 1 to length(m)– length(n)
        match = 0
        For j = 1 to length(n)
            if (n[j] = m[i+j–1])
                match = match + 1
        If match > length(n)* 0.95
            print 'i'
```

- variables with assignments: i, j and match store 3 integers
- we see from length(n) and n[j] assignment that n is a vector with length(n)
- m is a vector with length(m)

Memory: O(n+m+3) = O(n+m)

```
MatchMatrix(n, m):
    For i = 1 to length(n)
        For j = 1 to length(m)
            if (n[i] = m[j])
                matrix[i,j] = 1
            else
                matrix[i,j] = 0
```

- variables with assignments: i, j and matrix store integers
- i and j store single integers, but matrix stores up to n rows and m columns
- matrix[i,j], where i = 1..n and j = 1..m

Memory: O(n*m+2) = O(n*m)

Sort Example

Sorting Problem:
Sort a list of integers.

Input: A list of n distinct integers $\mathbf{a} = (a_1, a_2, \ldots, a_n)$.

Output: Sorted list of integers, that is, a reordering $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ of integers from $\mathbf{a}$ such that $b_1 < b_2 < \cdots < b_n$.

```
SelectionSort( a, n )
1    For i ← 1 to n - 1
2        j ← IndexofMin( a, i, n )
3        Swap elements ai and aj
4    return a

IndexofMin( array, first, last )
1    index ← first
2    for k ← first + 1 to last
3        if arrayk < arrayindex
4            index ← k
5    return index
```

Find minimum (1..n)
Swap with first
Find minimum (2..n)
Swap with second
etc

$(7, 92, 87, 1, 4, 3, 2, 6)$

$(1, 92, 87, 7, 4, 3, 2, 6)$

$(1, 2, 87, 7, 4, 3, 92, 6)$

$(1, 2, 3, 7, 4, 87, 92, 6)$

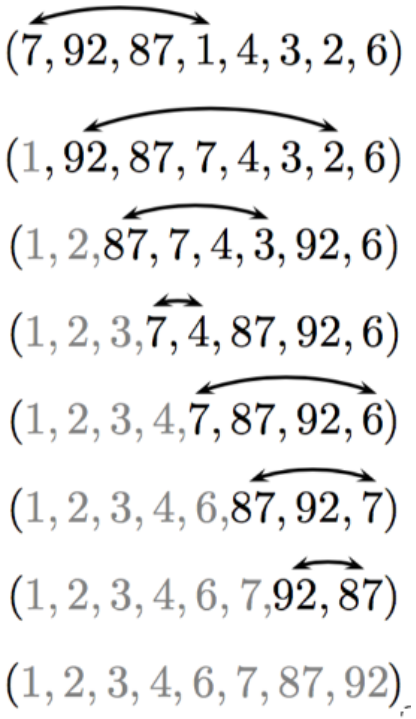
$(1, 2, 3, 4, 7, 87, 92, 6)$

$(1, 2, 3, 4, 6, 87, 92, 7)$

$(1, 2, 3, 4, 6, 7, 92, 87)$

$(1, 2, 3, 4, 6, 7, 87, 92)$

```
SelectionSort( a, n )
1     For i ←1 to n - 1
2           j ← IndexofMin( a, i, n )
3           Swap elements ai and aj
4     return a

IndexofMin( array, first, last )
1     index ← first
2     for k ← first + 1 to last
3           if arrayk < arrayindex
4                 index ← k
5     return index
```



Let's measure efficiency of SELECTIONSORT

- It makes *n-1* iterations in the for loop
- It analyzes *n-i+1* elements in iteration *i*
- The approximate number of operations performed is:
- *n+(n-1)+(n-2)+…+2+1 = 1 + 2 + … +n = n(n+1)/2.*

What is it's Big-O notation of this iterative algorithm?
$= n^2/2+n/2 = O(n^2)$
What is the memory requirement: O(n)+constant

Big O notation examples

Fibonnaci sequence: F(n)

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

n =	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Fn =	0	1	1	2	3	5	8	13	21	34

O(n)

```
FunctionMin(A_n):
A_1 = min
For i = 2 to n
    if A_i < min
        min = A_i
print 'min'
```

O(n)

```
Fibonnaci2(n):
F_1 = 1
F_2 = 1
For i = 3 to n
    F_i = F_i-1 + F_i-2
return(F_n)
```

$O(2^n)$

```
Fibonnaci(n):
if number <= 1
    return number
else
    return Fibonnaci(n-2) + Fibonnaci(n-1)
```

recursion

F(4) = F(3) + F(2)
= [F(2) + F(1)] + [F(1) + F(0)]
= [F(1) + F(0) + F(1)] + [F(1) + F(0)]

What about memory?
O(1) or O(n)

Exercises

1. What is the time and memory complexity of the following algorithms?

```
FunctionFactors(n):
    For i = 1 to n
        x = n*n
        print ‘x’

FunctionCombination2(n):
    For i = 1 to n-1
        For j = i to n
            print ‘i, j’

FunctionCombination3(n, m):
    For i = 1 to n
        For j = 1 to m
            print ‘i, j’
        For k = 1 to m
            print ‘i, k’
```

```
StringSearch(n, m):
    For i = 1 to length(m)- length(n)
        match = 0
        For j = 1 to length(n)
            if (n[j] = m[i+j-1])
                match = match + 1
        If match > length(n)* 0.95
            print 'i'

MatchMatrix(n, m):
    For i = 1 to length(n)
        For j = 1 to length(m)
            if (n[i] = m[j])
                matrix[i,j] = 1
            else
                matrix[i,j] = 0
```

2. Which of the following types of algorithms are inherently approximate? Which are stochastic?

exhaustive search, branch and bound, dynamic programming, greedy, randomization

3. Why would you use branch and bound over exhaustive?