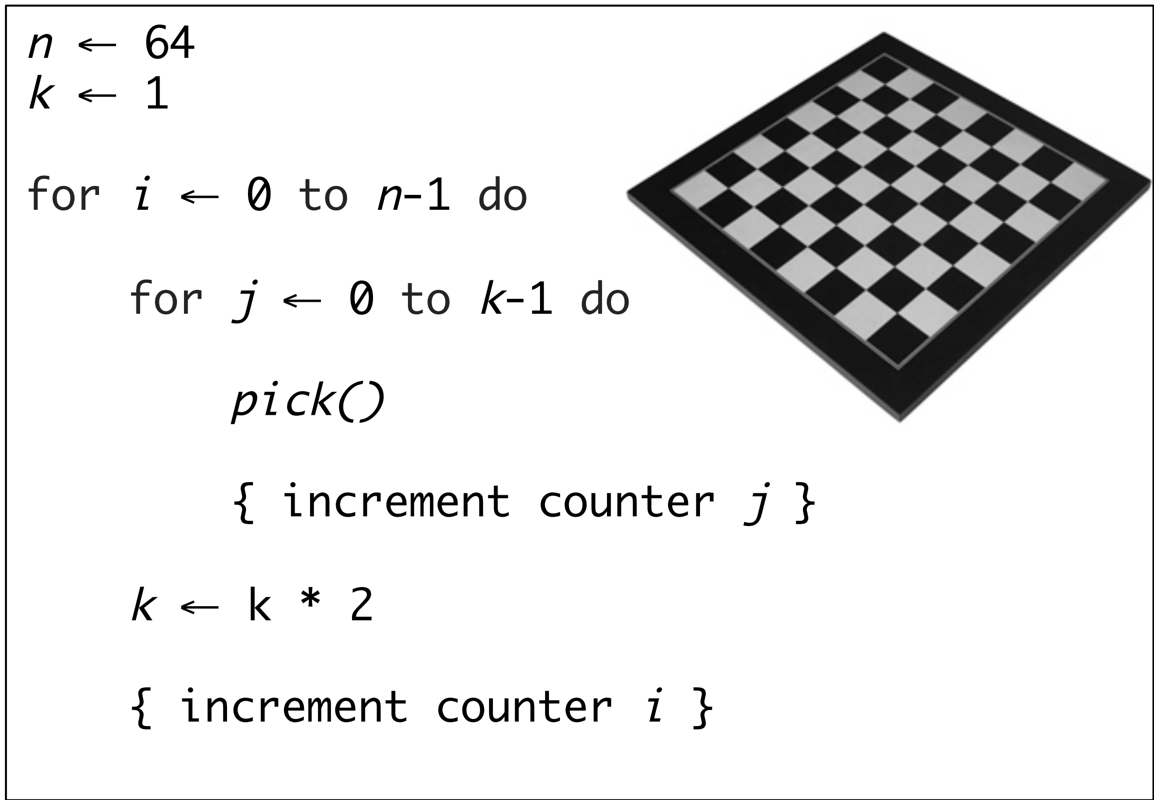
**Question 12**

Consider the following pseudo-code. Which of the following statements gives a fairly accurate indication of the number of times the underlined operation executes during the running of this method? The function read() is known to take constant time.

```
Algorithm g(n):
    Input: an integer n
    Output: an integer k
k ← 1
for i ← 0 to n-1 do
    for j ← 0 to k-1 do
        read()
    k ← k * 2
return k
```

A) $2^n$ times
B) $n^3$ times
C) $n^2$ times
D) $n \log n$ times
E) $n(n+1)/2$ times



# Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudo-code
- Largely independent from the programming language
- Exact definition not important (later we will see why)
- Assumed to take a constant amount of time
  - Assigning a value to a variable
  - Calling a method
  - Performing an arithmetic operation
  - Evaluating a conditional
  - Indexing into an array
  - Following an object reference
  - Returning from a method

**Question 12**

Consider the following pseudo-code. Which of the following statements gives a fairly accurate indication of the number of times the underlined operation executes during the running of this method? The function read() is known to take constant time.

```
Algorithm g(n):
    Input: an integer n
    Output: an integer k
k ← 1                           1
for i ← 0 to n-1 do             1 + 3n
    for j ← 0 to k-1 do         n + 3k but what is k in terms of n?
        read()                  1 + 2 + 4 + 8 + 2^4 + 2^5 + … + 2^(n-1)
    k ← k * 2                   2n
return k                        1
```

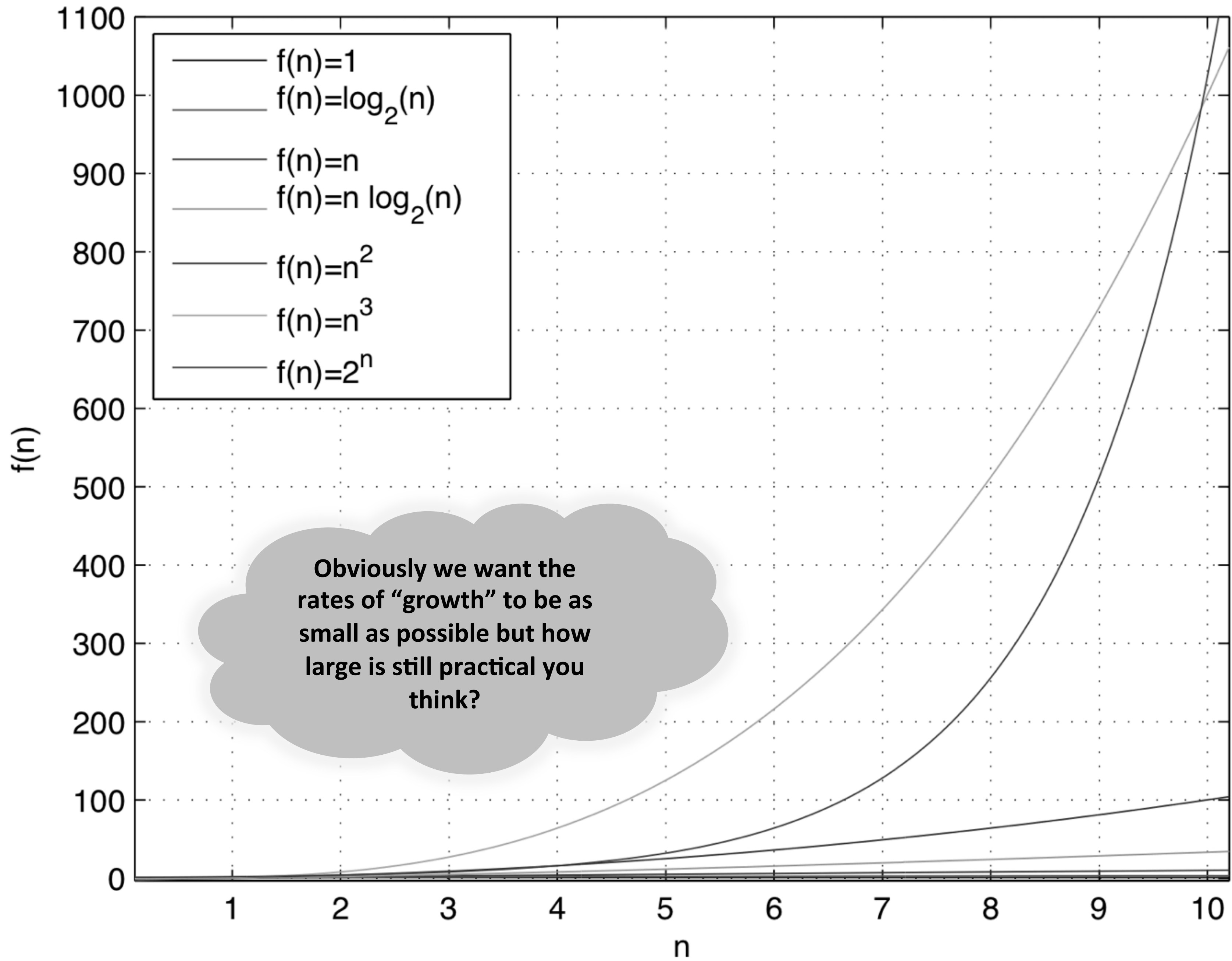
A) $2^n$ times

B) $n^3$ times

C) $n^2$ times

D) $n \log n$ times

E) $n(n+1)/2$ times


f(n)=1
f(n)=log2(n)
f(n)=n
f(n)=n log2(n)
f(n)=n2
f(n)=n3
f(n)=2n
f(n)
n
Obviously we want the rates of "growth" to be as small as possible but how large is still practical you think?

# Analysis Process

- The asymptotic analysis of an algorithm determines the running time in *big-Oh notation*

- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size - f($n$)
  - We express this function with big-Oh notation

# Big-Oh Rules

- If $f(n)$ is a polynomial of degree $d$, then $f(n)$ is $O(n^d)$, i.e.,
    - Drop lower-order terms
    - Drop constant factors (coefficients)
- Use the smallest possible class of functions
    - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
    - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

**Question 13**

Consider the following pseudo-code. What is the most accurate big-Oh statement about the time complexity of the algorithm? (The function read() is known to take constant time.)

```
Algorithm g(n):
    Input: an integer n
    Output: an integer k
k ← 1                               1
for i ← 0 to n-1 do                 1 + 3n
    for j ← 0 to k-1 do             n + 3k but what is k in terms of n?
        read()                      1 + 2 + 4 + 8 + 2^4 + 2^5 + … + 2^(n-1)
    k ← k * 2                       2n
return k                            1
```

A) $O(2^n)$
B) $O(3 \cdot 2^n)$
C) $O(3 \cdot 2^n + 4n)$
…

## Question 14

a. Draw a partial Huffman tree for a standard 7-bit ASCII, specifically include the following characters (note the character at the appropriate node)

| Dec | Oct | Hex | 8-bit bin | Char |
|---|---|---|---|---|
| 065 | 101 | 041 | 01000001 | A |
| 066 | 102 | 042 | 01000010 | B |
| 067 | 103 | 043 | 01000011 | C |
| 068 | 104 | 044 | 01000100 | D |
| 069 | 105 | 045 | 01000101 | E |
| 070 | 106 | 046 | 01000110 | F |
| 071 | 107 | 047 | 01000111 | G |
| 072 | 110 | 048 | 01001000 | H |

b. Now, assume messages we wish to encode only consist of these 8 characters. In fact, every second character in messages is a vowel (A and E), every other is a consonant (all others). You can assume that within each group characters occur evenly. Generate an optimal Huffman tree for this problem.

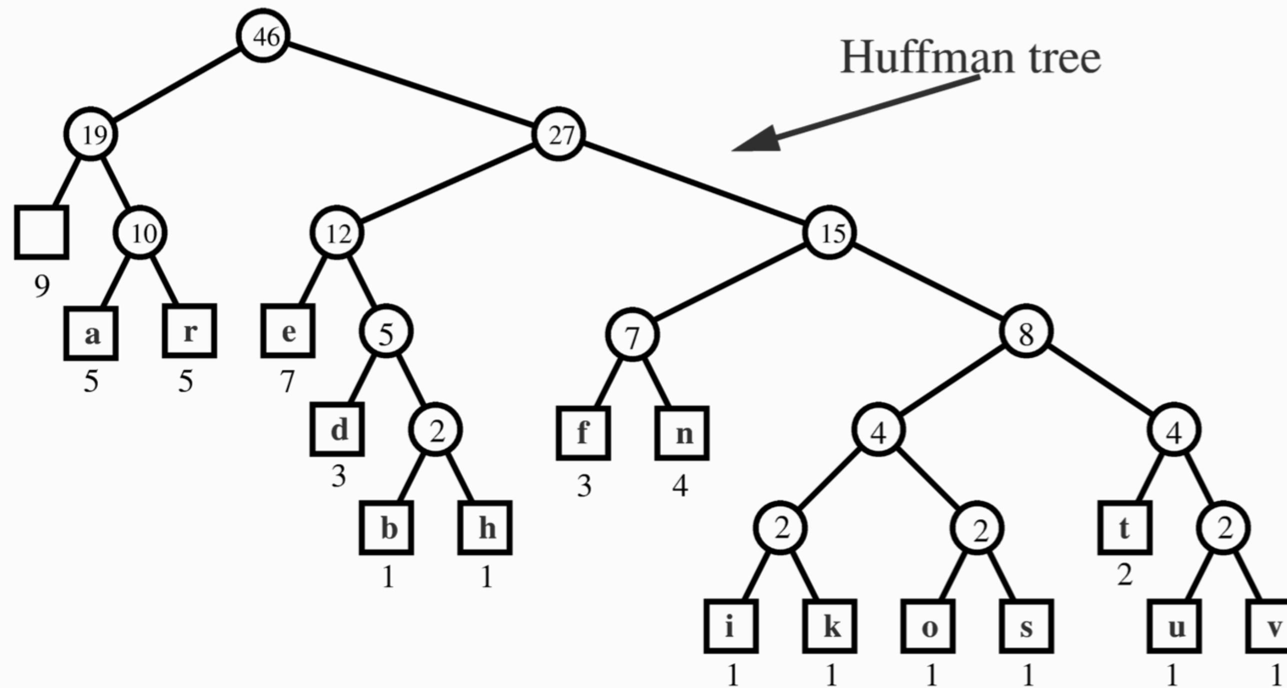
c. Use the Huffman tree from b to encode the message "BADABADABEGA". Specify how many bits less this message requires than if the Huffman tree from a was used.

# Huffman's Algorithm

- Given a string $X$, Huffman's algorithm constructs a prefix code that minimizes the size of the encoding of $X$
- It runs in time $O(n + d \log d)$, where $n$ is the size of $X$ and $d$ is the number of distinct characters of $X$
- A heap-based priority queue is used as an auxiliary structure hence $O(\log d)$ inside loop

**Algorithm** *HuffmanEncoding*($X$)
    **Input** string $X$ of size $n$
    **Output** optimal encoding trie for $X$
    $C \leftarrow$ *distinctCharacters*($X$)
    *computeFrequencies*($C$, $X$)
    $Q \leftarrow$ new priority queue
    **for all** $c \in C$
        $T \leftarrow$ new single-node tree storing $c$
        *Q.insert*(*getFrequency*($c$), $T$)
    **while** *Q.size*() > 1
        $f_1 \leftarrow$ *Q.minKey*()
        $T_1 \leftarrow$ *Q.removeMin*()
        $f_2 \leftarrow$ *Q.minKey*()
        $T_2 \leftarrow$ *Q.removeMin*()
        $T \leftarrow$ *join*($T_1$, $T_2$)
        *Q.insert*($f_1 + f_2$, $T$)
    **return** *Q.removeMin*()

# Huffman Tree Example




| Character | | a | b | d | e | f | h | i | k | n | o | r | s | t | u | v |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frequency | 9 | 5 | 1 | 3 | 7 | 3 | 1 | 1 | 1 | 4 | 1 | 5 | 1 | 2 | 1 | 1 |

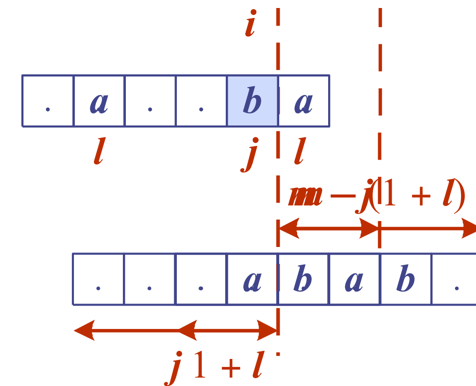
## Question 15

a. Search for “GCAGAGAG” in the text “GCATCGCAGCAGAGAGT”. Trace the comparisons used by Boyer-Moore.

b. Search for “ACGACGA” in the text “GCACCGCACGACGGACGACGAT”. Trace the comparisons used by Knuth-Morris-Pratt.

Case 2: $1 + l \leq j$ *(character jump)*

# Boyer-Moore




Search for “GCAGAGAG”
m = 8

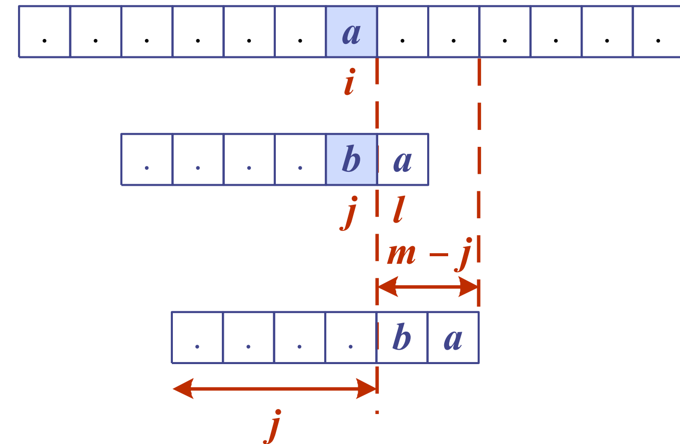
| c | A | C | G | T |
|---|---|---|---|---|
| L(c) | 6 | 1 | 7 | -1 |

```
Text :  GCATCGCAGCAGAGAGT
        GCAGAGAG          j=7  L(‘A’)=6
         GCAGAGAG         j=5  L(‘C’)=1
             GCAGAGAG     j=7  L(‘A’)=6
              GCAGAGAG    j=3  L(‘C’)=1
                GCAGAGAG  match!
```

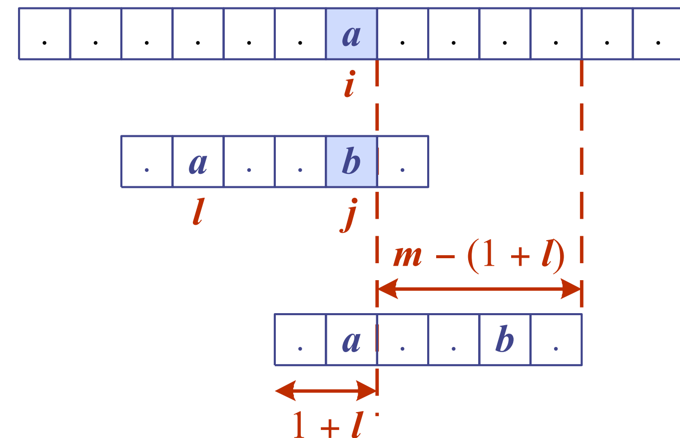
# The Boyer-Moore Algorithm

*Algorithm BoyerMooreMatch*($T$, $P$, $\Sigma$)
  $L \leftarrow$ *lastOccurenceFunction*($P$, $\Sigma$)
  $i \leftarrow m - 1$ { index for T, starts one pattern in - 1 }
  $j \leftarrow m - 1$ { index for P, starts at last element in P }
  **repeat**
    **if** $T[i] = P[j]$
      **if** $j = 0$
        **return** $i$
      **else**
        $i \leftarrow i - 1$
        $j \leftarrow j - 1$
    **else**
      $l \leftarrow L[T[i]]$
      $i \leftarrow i + m - \min(j, 1 + l)$
      $j \leftarrow m - 1$
  **until** $i > n - 1$
  **return** $-1$ { no match }

Case 1: $j \leq 1 + l$ *(shift, looking glass)*




Case 2: $1 + l \leq j$ *(character jump)*



# Knuth-Morris-Pratt

```
Search for “ACGACGA”
m = 7
j    :  0 1 2 3 4 5 6
P[j] :  A C G A C G A
F(j) :  0 0 0 1 2 3 4

Text : GCACCGCACGACGGACGACGAT
       ACGACGA
        ACGACGA
         ACGACGA
           ACGACGA
            ..
              ACGACGA
                 ACGACGA
                    ACGACGA
```

# The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - $i$ increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$ because length of prefix can not exceed that of suffix)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { no match }
```

## Question 16

a. Beside dynamic programming, list three principal algorithm design patterns for solving search problems

b. Define sequence alignment as a dynamic programming problem.

c. Show a dynamic programming matrix and explain how DP uses it to store solutions to sub-problems of increasing complexity. Use the following sequence alignment problem as an example.

```
X=GTTCTCAG
Y=GTTTGCTTAG
```

# General dynami

- Applies to a problem tha
  require a lot of time prov
  - Simple sub-problems: the
    defined in terms of a few
    on.
  - Sub-problem optimality: the global optimum value can be defined in terms of optimal sub-problems
  - Sub-problem overlap: the sub-problems are not independent, but instead they overlap (hence, should be constructed bottom-up).

## Algorithmic approaches

**Brute-force algorithms** – Easy to program, slow to run

**Branch-and-bound** – Given some clever analysis, we can usually eliminate many possibilities from the search

**Heuristic algorithms** – By making assumptions about our problem, we can obtain a faster algorithm, though these assumptions may sometimes make us slower or less accurate

**Greedy algorithms** – As a heuristic, we make the "greedy" choice, i.e., do what currently looks best (even if it isn't long term)

**Dynamic programming** – Break a problem into smaller sub-problems and use solutions to the sub-problem to construct the solution of the larger one

# Finding the best "alignment" between two sequences: X & Y

- Define D(i,j) to be the distance (or difference in number of characters) between X[0..i] and Y[0..j]
- Allow for -1 as an index, so

  $D(-1,k) = k$ and $D(k,-1) = k$

  to indicate that a the empty string and a k-long string are different by k characters

# The recurrence relation

Define distance:

$$D(i,j) = \min \begin{bmatrix} D(i-1, j-1) + c(i,j) \\ D(i-1, j) + 1 \\ D(i, j-1) + 1 \end{bmatrix} \begin{matrix} \text{MATCH} \\ \text{GAP in Y} \\ \text{GAP in X} \end{matrix}$$

where $D(i, j)$ is the distance between sequences X[0..$i$-1] and Y[0..$j$-1]

$D(-1,-1)=0$ by definition (the distance between two empty strings)

MATCH

$c(i, j) = 0$ when characters are identical,

$c(i, j) = 1$ when characters are different

GAP

cost is "1"

Fill in $D(i, j)$ for all cells

| | | G | T | T | T | G | C | T | T | A | G |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |
| G | | | | | | | | | | | |
| T | | | | | | | | | | | |
| T | | | | | | | | | | | |
| C | | | | | | | | | | | |
| T | | | | | | | | | | | |
| C | | | | | | | | | | | |
| A | | | | | | | | | | | |
| G | | | | | | | | | | | |

$$D(i,j) = \min \begin{bmatrix} D(i-1, j-1) + c(i,j) \\ D(i-1, j) + 1 \\ D(i, j-1) + 1 \end{bmatrix}$$