Scientific Computation

Spring 2019

Lecture 5

Imperial College
London

27 January, 2019

Notes

- HW1 will be posted later today (around ~6pm)
- Today’s office hour is 10-11, 6M 20

Today

- Hash functions for IP addresses revisited
- Characterizing running times
- Gene sequences and hashing strings

Hash functions

Last time:

- **The IP address is four integers,** a_1,a_2,a_3,a_4
- **Randomly choose four arbitrary integer weights,** w_1,w_2,w_3,w_4

- **Is:** $i = \Sigma w_j a_j$ **a suitable hash function?**

Answer depends on: 1) how the weights are chosen and 2) if we can establish an "even" distribution of the p **expected Ips**

Hash functions

Last time:

- **The IP address is four integers,** a_1,a_2,a_3,a_4
- **Randomly choose four arbitrary integer weights,** w_1,w_2,w_3,w_4

- **Is:** $i = \Sigma w_j a_j$ **a suitable hash function?**

Answer depends on: 1) how the weights are chosen and 2) if we can establish an "even" distribution of the p **expected IPs**

- **We want to use memory efficiently and limit the range of** i. **Simultaneously, we place constraints on the weights to ensure even distribution**

- **If we choose** $i = h(a_j)= \Sigma w_j \, a_j \bmod p$ **with** p **an appropriately large prime and constrain weights to be in** $0<w_j<p$ **we can establish that the probability of a hash collision is** $1/p$ **and limit memory usage to** $O(n+p)$ **where** n **is the actual number of IP addresses stored**
- **We have defined a *family* hash functions**

Analyzing running time

- On a basic level, analyzing running time is a matter of counting
  - E.g. a additions, b assignments, c comparisons/iteration
  - Then, with n iterations, cost is $a + b + cn$
  - Number of operations may vary based on input, so often need to consider worst-case, best-case, and/or average cost
  - Example: linear search with size-N input
    - Best case: 1 iteration, (1 assignments,1 comparison, 1 addition)/iteration, 3 operations
    - On average, 3N/2 operations
    - Worst case: 3N

Analyzing running time

- **Initial analysis usually focuses on *asymptotic running time***

- **Three approaches**

1.) Cost, $C(N)$ is $O(f(N))$ if there is a constant a, and an integer, N_0, where for all $N>N_0$, $C(N)<=a\ f(N)$

This is "Big-O" notation and establishes an upper bound

- **Example: if** $C < 8N \log_2 N + 8N$ **(merge sort when N is not a power of 2)**
  - **Then** C **is** $O(N \log_2 N)$
    - **Why?** $\log_2 N + 1 < 2\log_2 N$ **if** $N > 2$, **can then choose** $a=16$

- **This provides an upper bound and describes the worst-case scenario**
- **If the worst-case scenario is close to the best-case scenario, no need for other approaches**

Analyzing running time

- **Initial analysis usually focuses on *asymptotic running time***

- **Three approaches**

2.) Cost, C(N) is Ω(f(N)) if there is a constant a, and an integer, N_0, where for all $N>N_0$, C(N)>=a f(N)

- **This provides a lower bound and describes the best-case scenario**

3) The cost is Θ(f(N)) if and only if it is Ω(f(N)) *and* O(f(N))

Analyzing running time

- **We typically don't need to work with these precise definitions**

- **When I ask for information about the running time, I typically expect the construction of an estimate of the form:**
- $C = a_1 f_1(N) + a_2 f_2(N) + a_3 f_3(N) + \dots$

where a_i **are positive integers and the functions are ordered so that:**

$$lim_{N \to \infty} (f_{i+1}/f_i) = 0$$

Then the *leading-order* term is $a_1 f_1(N)$

and we'll say that C **is** $O(f_1(N))$

(though this doesn't match the definition provided earlier)

Genetic code

- DNA is constructed from 4 nucleotides (or *bases*):
  - Adenine
  - Cytosine
  - Guanine
  - Thymine

  (RNA has Uracil in place of Thymine)

- Adenine bonds with Thymine and Guanine bonds with Cytosine

- So if one strand contains the sequence GCTTCA the other strand will contain CGAAGT in the corresponding location

- During cell division, each daughter cell gets one strand
  - And then the needed 2nd strand can be constructed so A's pair with T's and C's pair with G's

Genetic code

- DNA is constructed from 4 nucleotides (or *bases*):
  - Adenine
  - Cytosine
  - Guanine
  - Thymine

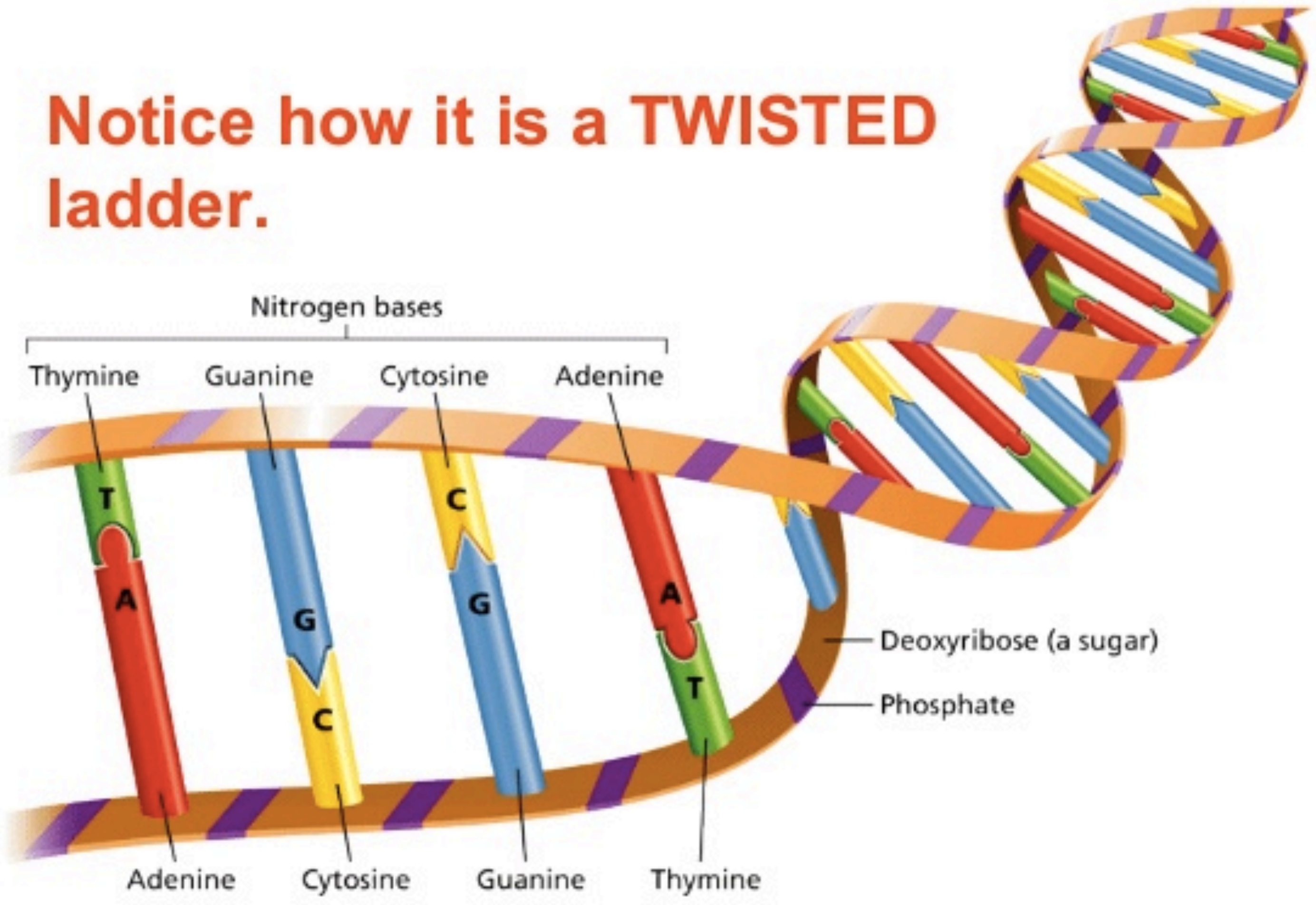
  (RNA has Uracil in place of Thymine)

- *Codons* consist of three DNA bases and contain code for synthesizing amino acids
  - Proteins are built from amino acids
  - 64 possible codons, but there are only 20 essential amino acids specified by DNA

- Gene sequencing involves:
  - Extracting the sequence of bases from DNA samples
  - Investigating the proteins or functions associated with codons and their sequences

Here is a DNA Molecule

Notice how it is a TWISTED ladder.



Genetic code

- In general, we want to find trends and patterns in gene sequences
- Examples:
    - ATG is a *start codon* and is present at the beginning of every DNA substring providing code for a protein (in eukaryotes)
    - Frequently occurring patterns point to important portions of a sequence and/or important substrings
    - The relative amount of cytosine and guanine can be use to find where in the sequence replication starts

Pattern search

- Problem setup:
  Specify a N-character sequence, *S*, and a M-character pattern, *P*
  Find all locations in *S* where *P* occurs

- Example:

  S = ATGTTGTACCGTATCGG
  P = GTA

  N=16, M=3

Pattern search

- Problem setup:
  Specify a N-character sequence, *S*, and a M-character pattern, *P*
  Find all locations in *S* where *P* occurs

- Example:

  S = ATGTTGTACCGTATCGG
  P = GTA

  N=16, M=3

- “Naive” approach:
  - Loop through *S* one character at a time
    - Check for matches with P one character at a time (less naïve: break the checking step after first mis-match)

Pattern search

- “Naive” approach:
  - Loop through *S* one character at a time
    - Check for matches with P one character at a time (breaking this check after first mis-match

```
#Set sequence
S = ‘ATGTTGTACCGTATCGG’
N = len(S)

#Set pattern for search
P = ‘GTA'
M = len(P)
```

Pattern search

- "Naive" approach:
    - Loop through *S* one character at a time
        - Check for matches with P one character at a time (breaking this check after first mis-match

```
#Set sequence
S = 'ATGTTGTACCGTATCGG'
N = len(S)

#Set pattern for search
P = 'GTA'
M = len(P)

for ind in range(0,N-M+1):
    matching=True
    #Compare sub-string to pattern
    for count,indp in enumerate(range(ind,ind+M)):
        if P[count] != S[indp]:
            matching=False
            break
    #Update list when match found
    if matching:
        imatch.append(ind)
        print("Match found, i=",ind)
```

Imperial College
London

Pattern search

- “Naive” approach:
  - Loop through *S* one character at a time
    - Check for matches with P one character at a time (breaking this check after first mis-match

```
#Set sequence
S = ‘ATGTTGTACCGTATCGG’
N = len(S)

#Set pattern for search
P = ‘GTA'
M = len(P)
```

```
In [15]: run naive_search
Match found, i= 5
Match found, i= 10
```

Imperial College
London

Pattern search

- “Naive” approach:
  - Loop through *S* one character at a time
    - Check for matches with P one character at a time (breaking this check after first mis-match

```
#Set sequence
S = ‘ATGTTGTACCGTATCGG’
N = len(S)

#Set pattern for search
P = ‘GTA'
M = len(P)
```

```
In [15]: run naive_search
Match found, i= 5
Match found, i= 10
```

- What is the cost?
  - Worst case, O(MN) operations when there are many “near-misses”
    - Can we mitigate the near-miss problem?

Imperial College
London

Pattern search

- “Naive” approach:
  - Loop through *S* one character at a time
    - Check for matches with P one character at a time (breaking this check after first mismatch
  - What is the cost?
    - Worst-case, O(MN) operations
    - (How) can we do better?
  - Binary search?
    - N $\log_2$N to sort
    - Then $\log_2$(N) for each search
    - But this requires storing N length-M strings/arrays
  - Hash table? Faster, but still with wasteful memory usage

Pattern search

- A (partial) solution:
  - Use a *rolling* hash function
    - Compute hash for pattern, *P*
    - Then apply function sequentially to each length-M substring in *S*
    - For a well-designed hash function, cost will be O(M + N)
    - And memory usage will also be O(M+N)

Pattern search

- **A (partial) solution:**
  - **Use a *rolling* hash function**
    - **Compute hash for pattern, *P***
    - **Then apply function sequentially to each length-M substring in *S***
    - **For a well-designed hash function, cost will be O(M + N)**
    - **And memory usage will also be O(M+N)**

- **What is a rolling hash function?**
  - **First, genetic sequences can be rewritten in base 4**
    - **A=0, C=1, G=2, T=3**
  - **A simplistic function – convert sequence from base 4 to base 10**
  - **Example: *S* = GCTAT =** 21303

$$H(S) = 2*4^4 + 1*4^3 + 3*4^2 + 0*4^1 + 3$$

  - **Or more generally, evaluate a M-1th-order polynomial for each length-M substring of *S***