Respostas do Teste Admissional

Candidato: Antonio Victor Campos Coelho

1. O que é um arquivo do tipo FASTQ e como posso verificar se um FASTQ é válido?

Um arquivo em formato FASTQ pode ser entendido como uma variante do formato FASTA. Esse formato une a sequência de nucleotídeos com uma medição da qualidade do sequenciamento (FASTA + Quality = FASTQ). Essa medição da qualidade é o tradicional *PHRED score (P)*, que representa a probabilidade de erro na determinação da base nitrogenada durante o sequenciamento. O *P* varia de 0 a 40, de maneira que `P=0` representa 100% de probabilidade de erro (portanto, qualidade baixíssima) e `P=40` representa 0.01% de erro (portanto, qualidade muito alta - 1 erro a cada 10 mil bases determinadas).

Para verificar se um FASTQ é válido, primeiro é preciso entender a especificação do formato. O formato consiste em quatro seções, e a informação de cada seção é contida em uma linha cada.

A primeira seção é o **cabeçalho**, que é similar ao formato FASTA: ele identifica a sequência e (opcionalmente) conter informações extras, tais como o nome do instrumento, código identificador da reação de sequenciamento ("*run*", "corrida"), a identificação da *flowcell* onde a reação de sequenciamento ocorre, faixa da *flowcell*, coordenadas dos *clusters* de sequenciamento na faixa da *flowcell*, etc. A diferença é que no FASTQ ele começa com `@`, enquanto o FASTA começa com `>`.

A segunda seção contém a **sequência** de bases nitrogenadas.

A terceira seção é uma linha que começa com o símbolo `+`. O cabeçalho pode vir repetido depois do símbolo.

A quarta e última seção codifica os valores de **qualidade de sequenciamento** de cada base nitrogenada. Cada valor de qualidade corresponde a um símbolo ASCII. O padrão mais difundido atualmente é o padrão Sanger, em que os códigos ASCII subtraídos de 33 conferem a qualidade. Exemplo: o símbolo `!` possui código ASCII 33. `33 - 33 = 0`, então `!` significa a mais baixa qualidade, enquanto o símbolo `I` tem código ASCII 73. `73 - 33 = 40`, então o símbolo `I` significa a mais alta qualidade.

Portanto, para verificar se um arquivo FASTQ é válido, deve-se verificar se os formatos das seções estão corretos. Uma primeira evidência é verificar a quantidade de linhas no arquivo. Explicando: se N é o número de sequências biológicas dentro do arquivo, então a quantidade de linhas ao todo no arquivo deve ser no máximo `4 * N + 1`, pois existem quatro seções e uma possível última linha vazia que encerra o arquivo. Em seguida, ferramentas de linha de comando como `grep` ou `awk` podem ser usadas para verificar, por exemplo:

- Se o número de linhas de cabeçalho (começando com `@`) está dentro do esperado;
- Se o cabeçalho tem pelo menos dois caracteres (o `@` e pelo menos outro identificador da sequência);
- Não há espaços entre o cabeçalho e a sequência;
- Todo cabeçalho tem identificação exclusiva;
- A sequência e as medições de qualidade devem ter o mesmo comprimento.

Alternativamente, já existem programas abertos com ferramentas para verificar se um FASTQ é válido, como por exemplo o fastQValidator ou fastq_utils.

2. Quais são as etapas mais comuns de um pipeline de bioinformática de NGS para DNAseq e quais ferramentas podem ser usadas em cada etapa?

De uma maneira geral, eu considero cinco etapas:

1- Checagem da qualidade das sequências

Essa etapa serve para avaliar se os arquivos FASTQ obtidos a partir de reações de NGS são válidas e possuem boa qualidade. Nessa etapa, uma ferramenta bastante usada é a `FASTQC`. Essa ferramenta fornece dados diagnósticos, como número de *reads* obtidas, qualidade por base, comprimento de *reads*, se há sequências duplicadas, se há a presença de sequências dos oligonucleotídeos adaptadores, entre outras informações. A depender dos resultados dessa checagem, podemos executar a segunda etapa ou pular diretamente para as demais.

2- *Trimming* ("aparamento")

O *trimming* é a eliminação de sequências de adaptadores de sequenciamento e bases com baixa qualidade. Caso seja executada, é recomendado que se repita a etapa 1 para verificar se houve melhora significativa na qualidade das *reads*. Algumas ferramentas usadas para essa etapa incluem Cutadapt, Trimmomatic e a ferramenta auxiliar Trim Galore!.

3- Alinhamento

Essa etapa serve para identificar (mapear) a qual região do genoma cada *read* corresponde. Essa etapa requer um arquivo FASTA contendo a sequência de referência do genoma. O *output* dessa etapa são arquivos formato *Sequence Alignment/Map* (SAM). As *reads* alinhadas nesses arquivos podem ser organizadas por coordenadas genômicas e indexadas ao passo que o arquivo é convertido para um formato binário (BAM), o que agiliza na anotação de regiões na etapa seguinte, além de serem mais compactos em tamanho.

Algumas ferramentas de alinhamento incluem o bwa e o bowtie2. A ferramenta `samtools` é utilizada para visualizar, escrever, editar e indexar arquivos SAM e BAM.

4- Remoção de sequências duplicadas devido a PCR e com baixa qualidade de mapeamento

Essa etapa é especialmente recomendada para diminuir erros de chamada de variantes. Como a chamada de variantes se dá com a atribuição de *scores* de confiabilidade baseado no número de vezes que determinada variante foi observada, se existem *reads* duplicadas, há o risco de se produzir um *score* de confiabilidade artificial. Por isso, cópias artificiais de sequências nas *reads* devem ser eliminadas. Normalmente, essas cópias acontecem durante processos de amplificação de PCR durante a preparação da biblioteca de sequenciamento. A ferramenta picard é normalmente usada para esse efeito.

Da mesma forma, sequências com baixa qualidade de mapeamento, ou seja, *reads* com alta probabilidade de que estejam em localização incorreta no genoma. Isso pode ser feito pela ferramenta `samtools`.

5- Chamada de variantes e anotação

Com os arquivos BAM organizados por coordenada, indexados e sem *reads* duplicadas ou com baixa qualidade de mapeamento, pode ser feita a etapa final de uma *pipeline* de NGS, que é a chamada de variantes. Nessa etapa, se identifica as posições do genoma do indivíduo que diferem do genoma de

referência. Com isso, doenças genéticas podem ser diagnosticadas, prognósticos de tratamentos de doenças como câncer podem ser obtidos, dentre outras finalidades.

Essa etapa pode ser feita com ferramentas como bcftools e freebayes. Com as variantes identificadas, é feito o processo de anotação, que é identificar qual a probabilidade de que as variantes encontradas terem impacto clínico no fenótipo do paciente. Isso é feito através da comparação com bancos de dados curados provenientes de estudos anteriores, que relacionam o tipo da variante (polimorfismo de único nucleotídeo ou SNP, inserções/deleções etc.) com o efeito molecular resultante. Essa etapa pode ser feita com ferramentas como a snpEff ou a Ensembl Variant Effect Predictor (vep)

O resultado dessa etapa são arquivos *Variant Call Format* (VCF) contendo uma variante por linha, identificada por colunas com o cromossomo e posição no cromossomo, bem como alelos de referência e o alelo observado, genótipo do paciente, qualidade da chamada, bem como a anotação contendo palavras chave que sumarizam o impacto daquela variante na função molecular do gene e consequentemente no fenótipo de interesse.

3. O que é o dogma central da biologia e como a bioinformática pode ser aplicada em cada um dos processos?

O dogma central da biologia molecular é uma explicação de como a informação biológica flui nos processos celulares, e foi postulado em 1958 por Francis Crick. O dogma central explica a transferência de informação sequencial que existe nos ácidos nucleicos até às proteínas. Em outras palavras, a informação biológica, representada por sequências de nucleotídeos, pode fluir de ácido nucleico para ácido nucleico:

- Durante a replicação (DNA -> DNA em todas as células, RNA -> RNA em vários tipos de vírus)
- Durante a transcrição (DNA -> RNA)
- Durante a transcrição reversa (RNA -> DNA, em retrovírus e retrotransposons celulares)

Mas assim que a informação é convertida para uma sequência de aminoácidos, ela não pode mais ser convertida em ácido nucleico de volta:

- Tradução (RNA -> proteína)

Sendo assim, a Bioinformática pode ser usada para entender a variação e quantificar o fluxo dessa informação biológica.

Em estudos de genômica (DNA-Seq), podemos identificar o número e o tipo de variações nos genomas para entender como elas estão associadas com doenças, elaborar prognóstico para tratamentos de saúde e entender a evolução humana. Para isso, utilizamos ferramentas de computação para alinhar o genoma examinado com genoma de referência para podermos mapear as variações existentes.

Em estudos de transcriptômica (RNA-Seq), podemos identificar o número e o tipo de moléculas de RNA transcritas num determinado tecido em determinada condição. Com isso, conseguimos entender como a célula responde a estímulos tanto no estado saudável quanto no estado patológico, resultando na descoberta de vias metabólicas que podem ser exploradas para o desenvolvimento de tratamentos, por exemplo.

Por fim, em estudos de proteômica e metabolômica podemos identificar o número e o tipo de proteínas produzidas por células e tecidos em determinada condição, sua estrutura, quais substratos elas consomem e quais produzem, quais modificações pós-tradução elas estão sofrendo em determinada condição. É a área mais complexa das ciências "ômicas" pois proteomas diferem de célula para célula e ao longo do tempo;

muitas vezes a informação obtida por estudos de transcriptômica não é diretamente correlacionável com estudos de proteômica, pois as células possuem vários mecanismos de regulação da tradução que fazem com que nem sempre a coleção de RNA mensageiro da célula naquele momento seja totalmente traduzida.

4. Considerando que o alinhamento de sequências biológicas é uma das atividades mais recorrentes e importantes na área de bioinformática. Comente sobre os algoritmos computacionais de alinhamentos mais utilizados e em que cenários uns são mais indicados do que outros?

De uma maneira geral, os algoritmos de alinhamento de sequências biológicas são divididos em algoritmos de alinhamento global, local e híbridos.

Os algoritmos de alinhamento global são adequados para verificar sequências que se pressupõe que são homólogas, como por exemplo, sequências dos mesmos genes investigados em pessoas diferentes, genes com as mesma função porém de espécies diferentes, etc. Ele é assim chamado porque o algoritmo tenta fazer um alinhamento usando toda a extensão das sequências em estudo, de ponta a ponta. Um dos primeiros algoritmos usado por ferramentas de alinhamento global foi o de Needleman-Wunsch. Mais recentemente, programas mais modernos como MUSCLE e MAFFT utilizam algoritmos iterativos enquanto outros utilizam modelos ocultos de Markov, como Clustal Omega.

Já os algoritmos de alinhamento local são adequados para identificar regiões com alto grau de similaridade nas sequências em comparação, desconsiderando regiões adjacentes. Por isso, eles são mais adequados para alinhar sequências mais divergentes ou com maior distância evolutiva, para identificar padrões de sequências conservadas, principalmente em sequências de proteínas. Uma ferramenta símbolo dos alinhamentos locais é o BLAST (*basic local alignment search tool*). O seu algoritmo é heurístico, de maneira que ele aplica atalhos computacionais para retornar vários resultados, mas que não são necessariamente os melhores resultados teóricos.

Em termos de análise de sequências obtidas por sequenciamento de nova geração (NGS), os algoritmos híbridos são mais utilizados, pois combinam características dos dois primeiros. Eles buscam o melhor possível alinhamento parcial de duas sequências. Isso é bastante útil para mapear sequências curtas (*short reads*) em genomas de referência. Comumente, ferramentas usadas para realizar alinhamento híbrido/mapeamento em estudos de DNA-Seq são o `bwa` e `bowtie2`, enquanto para RNA-Seq alguns dos mais usados são o STAR aligner e o HISAT2.

5. [Desafio Técnico] Reconstruir um cromossomo

Apresentação do Raciocínio

A minha abordagem para esse desafio foi utilizar o conceito do **gráfico de De Bruijn**. De modo breve, um gráfico de De Bruijn (Nicolaas de Bruijn, matemático holandês) é uma solução para um problema de *superstring*, tal como o desse desafio. De Bruijn demonstrou que era possível obter um gráfico circular contendo todas as possíveis *substrings* (que a partir de agora vou denominar de *k-mers*) de uma *superstring*. Cada *k-mer* representa um vértice e os *k-mers* são conectados por "caminhos" ou "bordas" caso eles sejam similares. Assim, De Bruijn percebeu, baseando-se nos trabalhos de Euler (Leonhard Euler, matemático suíço), que a *superstring* mais curta possível seria obtida caso visitássemos cada vértice do gráfico apenas uma vez e retornado ao vértice inicial. Esse processo é conhecido como ciclo Euleriano.

De Bruijn mostrou que existem n^k *k-mers* em um alfabeto contendo *n* símbolos. Por exemplo, nas sequências biológicas temos quatro bases: A, C, T e G. Portanto, há $4^3 = 64$ trinucleotídeos. Assim, segui a estratégia de obter todos os *k-mers* de uma lista de *reads* e com isso montar um gráfico de De Bruijn, de maneira que o gráfico contivesse cada *k-mer* apenas uma vez. Assim, ao visitar cada vértice apenas uma vez (porém sem retornar ao vértice inicial, o que é definido como "caminhada Euleriana" ou *Eulerian walk*), eu reconstituiria a sequência original.

Para isso, me baseando no artigo How to apply de Bruijn graphs to genome assembly e em algumas funções criadas pelo Dr. Deren Eaton e outros, escrevi funções no `Python3`, que estão salvas no arquivo `code/dna_assembly.py`.

Sumário das Funções desenvolvidas

Vou começar pela função que recebe o *input* de sequências curtas (*reads*) para a reconstrução da sequência original:

```
def assemble(input_file_path, output_file_path="output.txt", k=None):
    """Wrapper function for get_kmers, get_edges and get_superstring

    Args:
        file (file): Text file containing one sequencing read per line
         k (int, optional): Size of k-mer. Defaults to None. If None k will equal
the half of the read (rounded if read length is odd)

    Returns:
         str: The shortest superstring that contains all other strings represented
in the edges input
    """

    sequences = open_input(input_file_path)

    km = get_kmers(sequences, k=k)

    edges = get_edges(km)

    superstring = get_superstring(edges)
```

```
    export_output(superstring, output_file_path)
```

Os argumentos dessa função são o caminho do arquivo de input (`input_file_path`), o caminho do arquivo de output (`output_file_path`, cuja opção padrão é salvar um arquivo `output.txt` no diretório de trabalho ativo) e o comprimento desejado dos *k-mers*. O padrão dele é `None`. Se o `k` não for fornecido, é estabelecido como a metade (arredondada) do comprimento da menor *read* dentre as fornecidas.

Como a primeira parte da função é apenas armazenar as *reads* numa lista, que é feito pela função `open_input`, não me deterei nela. A lista com as sequências é então passada para a função `get_kmers`, que processará as *reads*:

```
def get_kmers(sequences, k=None):
    """Generate list of k-mers found in a list of sequences

    Args:
        sequences (list): Sequencing reads
         k (int, optional): Size of k-mer. Defaults to None. If None k will equal
the half of the read (rounded if read length is odd)

    Returns:
         dictionary: k-mer dictionary with counts of how many times it appeared on
the reads
    """

    if k:
        k = k
    else:
        k = len(min(sequences, key=len)) // 2

    res = []
    for seq in sequences:
        res.append(
            [
                seq[x:y]
                for x, y in combinations(range(len(seq) + 1), r=2)
                if len(seq[x:y]) == k
            ]
        )

    kmers = list(chain.from_iterable(res))

    return dict(Counter(kmers))
```

`get_kmers` retorna um dicionário, cujas chaves são os *k-mers* e seus valores são a quantidade de vezes que ele apareceu nas sequências. A contagem serve para se certificar de que cada *k-mer* de fato apareceu apenas uma vez. Caso contrário, outro valor de *k* pode ser escolhido. A função `assemble` não retorna esse valor dicionário para o usuário, mas a `get_kmers` pode ser usada independentemente caso seja desejado.

O dicionário de *k-mers* é então repassado para a função `get_edges`, que determinará a conexão entre cada *k-mer*, de maneira a criar a "caminhada Euleriana". De modo breve, as bordas são construídas assim: cada *(k - 1)-mer* é atribuído a um vértice do gráfico; caso um vértice possua um prefixo que equivale ao sufixo de outro, os dois vértices são conectados, formando um caminho (*edge*):

```
def get_edges(kmers):
    """Determine De Bruijn's edges from a dictionary of k-mers

    Args:
        kmers (dict): Dictionary of k-mers, output of get_kmers function

    Returns:
        set: A set containing tuples representing the edges of a De Bruijn's graph
    """

    combs = combinations(kmers, 2)

    edges = set()

    for km1, km2 in combs:
        if km1[1:] == km2[:-1]:
            edges.add((km1[:-1], km2[:-1]))
        if km1[:-1] == km2[1:]:
            edges.add((km2[:-1], km1[:-1]))

    return edges
```

O *output* da função `get_edges` é um conjunto contendo conexões diretas entre dois vértices do gráfico de De Bruijn. Por fim, esse conjunto é repassado à função `get_superstring`. Essa função organiza os "caminhos" num objeto `deque`, que funciona como uma fila de elementos. A função escolhe aleatoriamente um "caminho" do conjunto e o insere no `deque`.

Então, uma recursão é responsável por fazer a "caminhada Euleriana": enquanto o comprimento do `deque` for menor ou igual que o comprimento do conjunto de "caminhos", a função retirará um par de vértices (*nodes*) do conjunto; se o valor da esquerda (`node1`) for igual ao último vértice guardado no `deque`, o valor da direita (`node2`) é colocado no fim da fila; caso contrário, a função verifica se o valor da direita é igual ao primeiro da fila. Caso afirmativo, o valor da esquerda é inserido antes, se transformando no novo valor do começo da fila. Caso nenhuma dessas alternativas sejam satisfeitas, a função então pega um novo par do conjunto, até que a condição da recursão seja atingida.

```
def get_superstring(edges):
    """Generates a contig based on edges of a De Bruijn's graph via Eulerian path
walk

    Args:
          edges (set): A set containing tuples representing the edges of a De
Bruijn's graph. Output of get_edges function

    Returns:
```

```
        str: The shortest superstring that contains all other strings represented
in the edges input
    """
    deck = deque()

    node1, node2 = random.choice(tuple(edges))
    deck = deque([node1, node2])

    while len(deck) <= len(edges):

        for node1, node2 in edges:

            if node1 == deck[-1]:
                deck.append(node2)

            elif node2 == deck[0]:
                deck.appendleft(node1)

            else:
                continue

    path = list(deck)

    superstring = path[0] + "".join(map(lambda x: x[-1], path[1:]))
    return superstring
```

Por fim, a função reúne os vértices enfileirados numa lista e a *superstring* é produzida por meio de uma função anônima que concatena os *(k-1)-mers* eliminado a sobreposição de prefixos-sufixos, e a salva num arquivo de texto no diretório de trabalho ativo.

Teste das funções desenvolvidas

Para testar essas fórmulas, eu criei três fórmulas adicionais, `get_genome`, `get_read` e `simulate_reads`. A primeira fórmula serve para gerar uma sequência de DNA artificial aleatória de tamanho desejado. A `função simulate_reads` serve para gerar *reads* artificiais do tamanho e cobertura de sequenciamento desejados para representar as *strings* que formariam o *contig*.

Assim, gerei um "cromossomo" artificial de 1000 bases de comprimento e gerei *reads* artificiais com comprimento de 100 bases numa cobertura equivalente a 30X. Salvei as *reads* no arquivo `input_teste.txt`:

```
from dna_assembly import *
from Bio import pairwise2

genome = get_genome(genome_length=1000, seed=123)
reads = simulate_reads(genome, genome_length=1000, read_length=100, coverage=30,
seed=123)

with open("input_teste.txt", "w") as f:
    for element in reads:
        f.write(element + "\n")
```

Em seguida, carreguei as *reads* na função `assemble` e escolhi um valor de `k=55`. Esse valor indica que cada *read* gerou 46 *55-mers*, que foram então usados para criar a "caminhada Euleriana". O resultado contendo o genoma reconstituído foi salvo no arquivo `assembly.txt`.

```
assemble("input_teste.txt","assembly.txt", k=55)
```

Para comparar o resultado com a sequência original, carreguei o *output* e utilizei o método de alinhamento do módulo `Biopython`:

```
with open("assembly.txt", "r") as f:
    assembly = f.readline().strip()

pairwise2.align.globalxx(genome, assembly)
```

O resultado foi o seguinte (editado para facilitar a leitura):

```
[Alignment(

seqA='AATAGACTGATTAATACATTGAAGAGCTGGTGACCGTTGCCCCGTCTAGAGGC...CCTCCCAAGGTTTCATGCGT
CTCATAACCCTTATAAGAAACTCG',
seqB='AATAGACTGATTAATACATTGAAGAGCTGGTGACCGTTGCCCCGTCTAGAGGC...CCTCCCAAGGTTTCATGCGT
CTCATAACCCTTATAAGAAACTC-',
    score=999.0, start=0, end=1000)]
```

Como pode ser observado, a função reconstituiu a sequência original quase que perfeitamente; o único erro foi na última base, a qual ficou ausente no *output* de teste. Provavelmente devido ao fato dos vértices serem representados na forma *(k-1)-mer*. Preciso realizar mais testes até descobrir a maneira de corrigir esse erro.

O código com o teste está no arquivo `code/test_assembly.py`.

Discussão e Conclusão

A abordagem de gráficos de De Bruijn se mostrou eficaz. De fato, ferramentas abertas como o `minia` também utilizam dessa abordagem.

Acredito que, embora não foram 100% perfeitas, as funções que adaptei/desenvolvi não consomem muitos recursos computacionais e são menos complexas do que os códigos de outros desenvolvedores que se debruçaram no problema das *superstrings* que encontrei durante minhas leituras para decidir uma abordagem. Por exemplo, enquanto outros desenvolvedores criaram classes para representar o gráfico, eu apenas utilizei as classes já disponíveis no Python (listas, dicionários, conjuntos, `deque`). Outros utilizavam vários *loops* aninhados, que atrapalhavam a leitura e entendimento do código (por exemplo essa solução no fórum StackOverflow), enquanto eu utilizei poderosas ferramentas de iteração já disponíveis no `Python` (o módulo `collections.itertools`) para simplificar o código. No entanto, reforço que essas funções servem

apenas como exercício, pois sequências genômicas reais são mais "problemáticas": ocorrem erros de sequenciamento e possuem sequências repetitivas, por exemplo.

6. [Desafio Técnico] Pipeline de DNASeq

As respostas para esse desafio estão no arquivo `output/QUESTION.TXT`. O código utilizado está no arquivo `code/code.sh`.

Referências (*links* citados)

FastQValidator - Genome Analysis Wiki

nunofonseca/fastq_utils

Cutadapt 3.2 documentation

USADELLAB.org - Trimmomatic: A flexible read trimming tool for Illumina NGS data

Babraham Bioinformatics - Trim Galore!

Burrows-Wheeler Aligner

Bowtie 2: fast and sensitive read alignment

Samtools

Picard Tools - By Broad Institute

freebayes/freebayes

Home - SnpEff & SnpSift Documentation

VEP command line

MUSCLE | Multiple Sequence Alignment | EMBL-EBI

MAFFT - a multiple sequence alignment program

Clustal Omega | Multiple Sequence Alignment | EMBL-EBI

alexdobin/STAR

HISAT2

How to apply de Bruijn graphs to genome assembly

Eaton Lab | nb-10.2-de-Bruijn

GATB/minia

Eulerian path and circuit for undirected graph - GeeksforGeeks

Reply to More efficient algorithm for shortest superstring search