[Knu11, Chapter 7.2.1.2]

Generating Permutations

Sam Ruggerio

Outline

Permutations

Algorithm L

Algorithm P

Conclusion

Section 1

Permutations

What is a Permutation

- Given a (multi)set S a permutation is an ordered sequence of every item in S

What is a Permutation

- Given a (multi)set S a permutation is an ordered sequence of every item in S
  - A set is an unordered collection of distinct items
  - A multiset allows repeated items

What is a Permutation

- Given a (multi)set S a permutation is an ordered sequence of every item in S
  - A set is an unordered collection of distinct items
  - A multiset allows repeated items
- For n elements there are $n!$ possible permutations

What is a Permutation

- Given a (multi)set S a permutation is an ordered sequence of every item in S
  - A set is an unordered collection of distinct items
  - A multiset allows repeated items
- For n elements there are $n!$ possible permutations
  - n possibilities for the first item... $n-1$ for the second... so on

Enumerating Permutations

- No surprises here, algorithms to enumerate permutations are going to require $O(n!)$ time

Enumerating Permutations

- No surprises here, algorithms to enumerate permutations are going to require $O(n!)$ time
- It's nice if algorithms output permutations in either
  (a) Some sorted order (lexographic)
  (b) Minimal change between permutations (Gray)

Enumerating Permutations

- No surprises here, algorithms to enumerate permutations are going to require $O(n!)$ time
- It's nice if algorithms output permutations in either
  (a) Some sorted order (lexographic)
  (b) Minimal change between permutations (Gray)
- Our algorithm will be slow in it's entirety, but we want to minimize "delay" between outputs.
  - We want to achieve $O(1)$ delay between permutations

Section 2

Algorithm L

Lexographic Permutation

- A lexographic permutation of the multiset {1,2,2,3}:

1223	1232	1322	2123
2132	2213	2231	2312
2321	3122	3212	3221

Lexographic Permutation

- A lexographic permutation of the multiset {1,2,2,3}:

1223	1232	1322	2123
2132	2213	2231	2312
2321	3122	3212	3221

- Let's assume we start a sequence $\{a_1, \ldots, a_n\}$ that is sorted, such that $a_1 \leq a_2 \leq \cdots \leq a_n$
- We also insert a sentinel a_0 that's smaller than everything.

Algorithm L

AlgorithmL$(S[a_0, \ldots, a_n])$:

```
j ← n − 1
while j > 0:
    PRINT(S)
    j ← n − 1
    while S[j] ≥ S[j + 1]:
        decrement j
    if j = 0:
        continue
    l ← n
    while S[j] ≥ S[l]
        decrement l
    SWAP(S[j], S[l])
    REVERSE(S, j + 1, n)
```

ALGORITHML$(S[a_0, \ldots, a_n])$:

```
j ← n − 1
while j > 0:
    PRINT(S)
    j ← n − 1
    while S[j] ≥ S[j + 1]:
        decrement j
    if j = 0:
        continue
    l ← n
    while S[j] ≥ S[l]
        decrement l
    SWAP(S[j], S[l])
    REVERSE(S, j + 1, n)
```

- If $j = 0$, we've reached our sentinel value, and there's nothing left to permute

AlgorithmL$(S[a_0, \ldots, a_n])$:

$j \leftarrow n - 1$
while $j > 0$:
    PRINT(S)
    $j \leftarrow n - 1$
    while $S[j] \geq S[j+1]$:
        decrement j
    if $j = 0$:
        continue
    $l \leftarrow n$
    while $S[j] \geq S[l]$
        decrement l
    SWAP$(S[j], S[l])$
    REVERSE$(S, j+1, n)$

- If $j = 0$, we've reached our sentinel value, and there's nothing left to permute
- Otherwise, each iteration of this while loop will print out a permutation.

AlgorithmL$(S[a_0, \ldots, a_n])$:

```
j ← n − 1
while j > 0:
    PRINT(S)
    j ← n − 1
    while S[j] ≥ S[j + 1]:
        decrement j
    if j = 0:
        continue
    l ← n
    while S[j] ≥ S[l]
        decrement l
    SWAP(S[j], S[l])
    REVERSE(S, j + 1, n)
```

- We find the largest j such that a_j can be increased

Algorithm L$(S[a_0, \ldots, a_n])$:

$j \leftarrow n - 1$
while $j > 0$:
  PRINT(S)
  $j \leftarrow n - 1$
  while $S[j] \geq S[j+1]$:
    decrement j
  if $j = 0$:
    continue
  $l \leftarrow n$
  while $S[j] \geq S[l]$
    decrement l
  SWAP$(S[j], S[l])$
  REVERSE$(S, j+1, n)$

- We find the largest j such that a_j can be increased
- Then, we find the smallest amount we can increase a_j by (the search for l).

ALGORITHML$(S[a_0, \ldots, a_n])$:

$j \leftarrow n - 1$
while $j > 0$:
    PRINT(S)
    $j \leftarrow n - 1$
    while $S[j] \geq S[j+1]$:
        decrement j
    if $j = 0$:
        continue
    $l \leftarrow n$
    while $S[j] \geq S[l]$
        decrement l
    SWAP$(S[j], S[l])$
    REVERSE$(S, j+1, n)$

- We've generated the prefix $a_1, \ldots, a_j$, but the second half is now $a_n, \ldots, a_{j+1}$, so we reverse it.

Questions?

Questions!

- What is the delay between permutations?
- What is (crudely) the runtime of this algorithm?
- If elements in S are distinct, how often does the decrement j step *not* run?

Section 3

Algorithm P

Gray Permutation

- We want to generate permutations in a way so that only two adjacent elements swap at every iteration

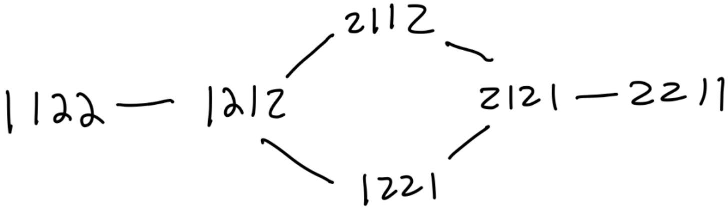
Gray Permutation

- We want to generate permutations in a way so that only two adjacent elements swap at every iteration
- This isn't guaranteed to happen in a multiset.

Gray Permutation

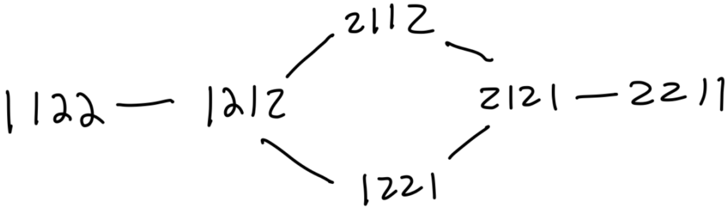
- We want to generate permutations in a way so that only two adjacent elements swap at every iteration
- This isn't guaranteed to happen in a multiset.
  - Consider a graph of permutations, where there are edges between adjacent swaps
  - We want to find a Hamiltonian path, but multisets can generate graphs where there are none!

Gray Paths



- No path that covers all nodes

Gray Paths



- No path that covers all nodes
- Luckily, permuting distinct sets is usually what happens most often in practice

Intuition

- Consider trying to find the permutations when $n = 4$: $\{1, 2, 3, 4\}$

Intuition

- Consider trying to find the permutations when $n = 4$: $\{1, 2, 3, 4\}$
- What if you took the permutations of $n = 3$, and inserted 4 into every position?

Intuition

- Consider trying to find the permutations when $n = 4$: $\{1, 2, 3, 4\}$
- What if you took the permutations of $n = 3$, and inserted 4 into every position?

$$123 \quad 132 \quad 312 \quad 321 \quad 231 \quad 213$$

Intuition

- Consider trying to find the permutations when $n = 4$: $\{1, 2, 3, 4\}$
- What if you took the permutations of $n = 3$, and inserted 4 into every position?

$$123 \quad 132 \quad 312 \quad 321 \quad 231 \quad 213$$

- Inserting 4's in a snaking pattern by column, you have your sequence of permutations

$$1234 \quad 1324 \quad 3124 \quad 3214 \quad 2314 \quad 2134$$

Intuition

- Consider trying to find the permutations when $n = 4$: $\{1, 2, 3, 4\}$
- What if you took the permutations of $n = 3$, and inserted 4 into every position?

123	132	312	321	231	213

- Inserting 4's in a snaking pattern by column, you have your sequence of permutations

1234	1324	3124	3214	2314	2134
1243	1342	3142	3241	2341	2143

Intuition

- Consider trying to find the permutations when $n = 4$: $\{1, 2, 3, 4\}$
- What if you took the permutations of $n = 3$, and inserted 4 into every position?

$$123 \quad 132 \quad 312 \quad 321 \quad 231 \quad 213$$

- Inserting 4's in a snaking pattern by column, you have your sequence of permutations

1234	1324	3124	3214	2314	2134
1243	1342	3142	3241	2341	2143
1423	1432	3412	3421	2431	2413
4123	4132	4312	4321	4231	4213

Algorithm P

$\underline{\textsc{AlgorithmP}(S[a_0, \ldots, a_n])}$:

$C[1..n] \leftarrow 0, O[1..n] \leftarrow 1$
while TRUE:
  PRINT(S)
  $j \leftarrow n, s \leftarrow 0$
  A: $q \leftarrow C[j] + O[j]$
    if $q < 0$: goto **D**
    if $q = j$: goto **B**
    SWAP$(S, j - C[j] + s, j - q + s)$
    $C[j] \leftarrow q$
    continue
  B: **if** $j = 1$:
      break
    $s \leftarrow s + 1$
  D: $O[j] \leftarrow -O[j], j \leftarrow j - 1$
    goto **A**

ALGORITHMP$(S[a_0, \ldots, a_n])$:

$C[1..n] \leftarrow 0, O[1..n] \leftarrow 1$
while TRUE:
  PRINT(S)
  $j \leftarrow n, s \leftarrow 0$
  A: $q \leftarrow C[j] + O[j]$
    if $q < 0$: goto **D**
    if $q = j$: goto **B**
    SWAP$(S, j - C[j] + s, j - q + s)$
    $C[j] \leftarrow q$
    continue
  B: **if** $j = 1$:
      break
    $s \leftarrow s + 1$
  D: $O[j] \leftarrow -O[j], j \leftarrow j - 1$
    goto **A**

- We initialize C, which tracks inversions, i.e. the distance a_k is from k in later iterations

AlgorithmP($S[a_0, \ldots, a_n]$):

$C[1..n] \leftarrow 0, O[1..n] \leftarrow 1$
while TRUE:
  PRINT(S)
  $j \leftarrow n, s \leftarrow 0$
  A: $q \leftarrow C[j] + O[j]$
    if $q < 0$: goto **D**
    if $q = j$: goto **B**
    SWAP($S, j - C[j] + s, j - q + s$)
    $C[j] \leftarrow q$
    continue
  B: **if** $j = 1$:
      break
    $s \leftarrow s + 1$
  D: $O[j] \leftarrow -O[j], j \leftarrow j - 1$
    goto **A**

- We initialize C, which tracks inversions, i.e. the distance a_k is from k in later iterations
- We initialize O which tracks which direction values in C have changed (left or right)

AlgorithmP$(S[a_0, \ldots, a_n])$:

$C[1..n] \leftarrow 0, O[1..n] \leftarrow 1$
while TRUE:
  PRINT(S)
  $j \leftarrow n, s \leftarrow 0$
  A: $q \leftarrow C[j] + O[j]$
    if $q < 0$: goto **D**
    if $q = j$: goto **B**
    SWAP$(S, j - C[j] + s, j - q + s)$
    $C[j] \leftarrow q$
    continue
  B: **if** $j = 1$:
      break
    $s \leftarrow s + 1$
  D: $O[j] \leftarrow -O[j], j \leftarrow j - 1$
    goto **A**

- We initialize C, which tracks inversions, i.e. the distance a_k is from k in later iterations
- We initialize O which tracks which direction values in C have changed (left or right)
- Every iteration, we print out a permutation

AlgorithmP$(S[a_0, \ldots, a_n])$:

```
C[1..n] ← 0, O[1..n] ← 1
while TRUE:
    PRINT(S)
    j ← n, s ← 0
    A: q ← C[j] + O[j]
        if q < 0: goto D
        if q = j: goto B
        SWAP(S, j − C[j] + s, j − q + s)
        C[j] ← q
        continue
    B: if j = 1:
            break
        s ← s + 1
    D: O[j] ← −O[j], j ← j − 1
        goto A
```

- We track the coordinate j where $C[j]$ is about to change, such that $0 \leq C[j] < j$ for all j

AlgorithmP$(S[a_0, \ldots, a_n])$:

```
C[1..n] ← 0, O[1..n] ← 1
while True:
    print(S)
    j ← n, s ← 0
    A:  q ← C[j] + O[j]
        if q < 0: goto D
        if q = j: goto B
        swap(S, j − C[j] + s, j − q + s)
        C[j] ← q
        continue
    B:  if j = 1:
            break
        s ← s + 1
    D:  O[j] ← −O[j], j ← j − 1
        goto A
```

- We track the coordinate j where $C[j]$ is about to change, such that $0 \leq C[j] < j$ for all j
- s tracks the number of indices k such that $C[k] = k - 1$ where $k > j$

AlgorithmP$(S[a_0, \ldots, a_n])$:

$C[1..n] \leftarrow 0, O[1..n] \leftarrow 1$
while True:
  Print(S)
  $j \leftarrow n, s \leftarrow 0$
  A: $q \leftarrow C[j] + O[j]$
    if $q < 0$: goto **D**
    if $q = j$: goto **B**
    Swap$(S, j - C[j] + s, j - q + s)$
    $C[j] \leftarrow q$
    continue
  B: **if** $j = 1$:
      break
    $s \leftarrow s + 1$
  D: $O[j] \leftarrow -O[j], j \leftarrow j - 1$
    goto **A**

- We determine q from C and O. If q is less than 0, we switch directions at **D**. If $q = j$ we need to increase s and switch directions, if possible

ALGORITHMP$(S[a_0, \ldots, a_n])$:

$C[1..n] \leftarrow 0, O[1..n] \leftarrow 1$
while TRUE:
    PRINT(S)
    $j \leftarrow n, s \leftarrow 0$
    A: $q \leftarrow C[j] + O[j]$
        if $q < 0$: goto **D**
        if $q = j$: goto **B**
        SWAP$(S, j - C[j] + s, j - q + s)$
        $C[j] \leftarrow q$
        continue
    B: **if** $j = 1$:
            break
        $s \leftarrow s + 1$
    D: $O[j] \leftarrow -O[j], j \leftarrow j - 1$
        goto **A**

- We determine q from C and O. If q is less than 0, we switch directions at **D**. If $q = j$ we need to increase s and switch directions, if possible
- Otherwise, we swap the two relative locations within S and print out a permutation

AlgorithmP$(S[a_0, \ldots, a_n])$:

```
C[1..n] ← 0, O[1..n] ← 1
while TRUE:
    PRINT(S)
    j ← n, s ← 0
    A:  q ← C[j] + O[j]
        if q < 0: goto D
        if q = j: goto B
        SWAP(S, j − C[j] + s, j − q + s)
        C[j] ← q
        continue
    B:  if j = 1:
            break
        s ← s + 1
    D:  O[j] ← −O[j], j ← j − 1
        goto A
```

- If we need to increase s, we check if $j = 1$ and terminate, then we move to **D** to switch directions

AlgorithmP$(S[a_0, \ldots, a_n])$:

```
C[1..n] ← 0, O[1..n] ← 1
while TRUE:
    PRINT(S)
    j ← n, s ← 0
    A:  q ← C[j] + O[j]
        if q < 0: goto D
        if q = j: goto B
        SWAP(S, j − C[j] + s, j − q + s)
        C[j] ← q
        continue
    B:  if j = 1:
            break
        s ← s + 1
    D:  O[j] ← −O[j], j ← j − 1
        goto A
```

- If we need to increase s, we check if $j = 1$ and terminate, then we move to **D** to switch directions
- **D** switches the direction stored in O and decrements j and returns to **A** to try again

Questions?

Section 4

Conclusion

Other interesting algorithms from this chapter:

- Algorithm T: Extends Algorithm P to output an array of transitions, to reuse generating permutations in constant time

Other interesting algorithms from this chapter:

- Algorithm T: Extends Algorithm P to output an array of transitions, to reuse generating permutations in constant time
- Algorithm G: A generalized approach to producing permutations when given a group of subsets of S

Other interesting algorithms from this chapter:

- Algorithm T: Extends Algorithm P to output an array of transitions, to reuse generating permutations in constant time
- Algorithm G: A generalized approach to producing permutations when given a group of subsets of S
- Algorithm X: An extension to Algorithm L to generate permutations satisfying some conditions efficiently
  - It does this by maintaining a linked list of available elements
  - This is not Algorithm X from last week

A permutation on the ten decimal digits is simply a 10 digit decimal number in which all digits are distinct. Hence all we need to do is to produce all 10 digit numbers and select only those who digits are distinct. Isn't it wonderful how high speed computing saves us from the drudgery of thinking! We simply program $k + 1 \to k$ and examine the digits of k for undesirable equalities. This gives us the permutations in dictionary order too!
On second sober thought ... we do need to think of something else.

— D. H. LEHMER (1957)

Bibliography

Donald E. Knuth.
The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.
Addison-Wesley Professional, 2011.