**COS 460 – Algorithms (week 06)**

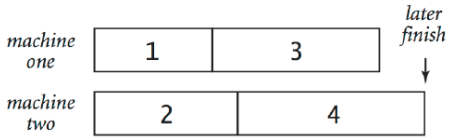
**Searching Algorithms Overview**

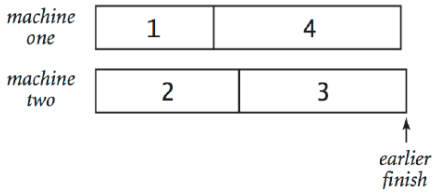
- **Exhaustive search**. Iterate through all elements of a search space.
- **Backtracking.** Systematic method for generating all solutions to a problem, by successively augmenting partial solutions.
- **Applicability.** Huge range of problems (include NP-hard ones).
- **Caveat.** Search space is typically exponential in size! Effectiveness is limited to relatively small instances.

**Enumerating**
**Enumerating Subsets**

**Scheduling (set partitioning).** Given $n$ jobs of varying length, divide among two machines to minimize the time (or, equivalently, difference between finish times) the last job finishes.

| job | length |
|---|---|
| 1 | 1.41 |
| 2 | 1.73 |
| 3 | 2.00 |
| 4 | 2.23 |




Enumerating subsets. Given $n$ items, enumerate all $2^n$ subsets.

- Count in binary from 0 to $2^n - 1$.
- Look at binary representation.

| *integer* | *binary code* | *machine one* | *machine two* |
|---|---|---|---|
| 0 | 0 0 0 0 | empty | 4 3 2 1 |
| 1 | 0 0 0 1 | 1 | 4 3 2 |
| 2 | 0 0 1 0 | 2 | 4 3 1 |
| 3 | 0 0 1 1 | 2 1 | 4 3 |
| 4 | 0 1 0 0 | 3 | 4 2 1 |
| 5 | 0 1 0 1 | 3 1 | 4 2 |
| 6 | 0 1 1 0 | 3 2 | 4 1 |
| 7 | 0 1 1 1 | 3 2 1 | 4 |
| 8 | 1 0 0 0 | 4 | 3 2 1 |
| 9 | 1 0 0 1 | 4 1 | 3 2 |
| 10 | 1 0 1 0 | 4 2 | 3 1 |
| 11 | 1 0 1 1 | 4 2 1 | 3 |
| 12 | 1 1 0 0 | 4 3 | 2 1 |
| 13 | 1 1 0 1 | 4 3 1 | 2 |
| 14 | 1 1 1 0 | 4 3 2 | 1 |
| 15 | 1 1 1 1 | 4 3 2 1 | empty |

```
int N = 1 << n;
for (int i = 0; i < N; i++)
 {
  int b = i;
  for(int j=0; j<n; j++)
   {cout << b%2; b /=2;}
  cout << endl;
 }
```

Output for $n = 4$

```
0000
1000
0100
1100
0010
1010
0110
1110
0001
1001
0101
1101
0011
1011
0111
1111
```

"Samuel Beckett"
*Quad*. Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.

| *code* | *subset* | *move* |
|---|---|---|
| 0 0 0 0 | *empty* | |
| 0 0 0 1 | 1 | enter 1 |
| 0 0 1 1 | 2 1 | enter 2 |
| 0 0 1 0 | 2 | exit 1 |
| 0 1 1 0 | 3 2 | enter 3 |
| 0 1 1 1 | 3 2 1 | enter 1 |
| 0 1 0 1 | 3 1 | exit 2 |
| 0 1 0 0 | 3 | exit 1 |
| 1 1 0 0 | 4 3 | enter 4 |
| 1 1 0 1 | 4 3 1 | enter 1 |
| 1 1 1 1 | 4 3 2 1 | enter 2 |
| 1 1 1 0 | 4 3 2 | exit 1 |
| 1 0 1 0 | 4 2 | exit 3 |
| 1 0 1 1 | 4 2 1 | enter 1 |
| 1 0 0 1 | 4 1 | exit 2 |
| 1 0 0 0 | 4 | exit 1 |

ruler function

```
void rule(int L, int R, int h)
{ int m=(L+R)/2;
  if(h>0)
   {p[m]=h;
    rule(L,m,h-1);
    rule(m,R,h-1);
   }
 }
```

For example, the call `rule(0,16,4)` yields the following contents of `p[]`:
`0121312141213121`

```
void moves(int n, bool enter)
{
 if (n == 0) return;
 moves(n-1, true);
 if (enter) cout << "enter " << n << endl;
 else cout << "exit " << n << endl;
 moves(n-1, false);
}
```

For example, the call `moves(4, true)` yields:

```
enter 1
enter 2
exit 1
enter 3
enter 1
exit 2
exit 1
enter 4
enter 1
enter 2
exit 1
exit 3
enter 1
exit 2
exit 1
```

```
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
```

stage directions for 3-actor play

`moves(3, true)`

```
enter 4
```

```
enter 1
enter 2
exit  1
exit  3
enter 1
exit  2
exit  1
```

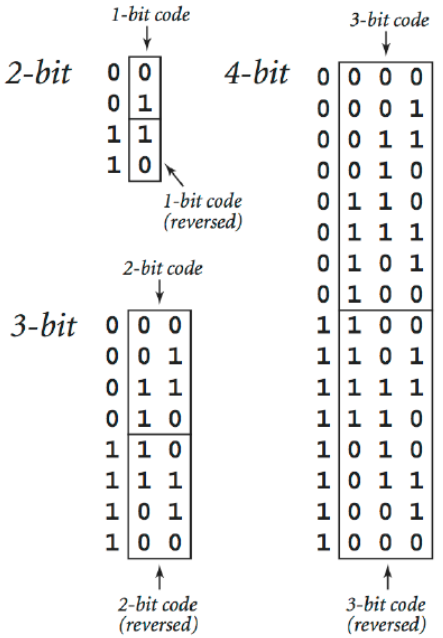
reverse stage directions for 3-actor play

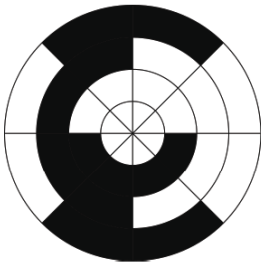
`moves(3, false)`

**Enumerating Subsets: Binary Reflected Gray Code**

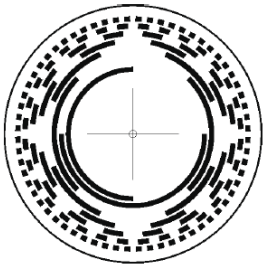
The *n*-bit code is:

- the (*n* – 1) bit code with a 0 prepended to each word, followed by
- the (*n* – 1) bit code in reverse order, with a 1 prepended to each word.



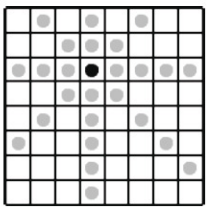



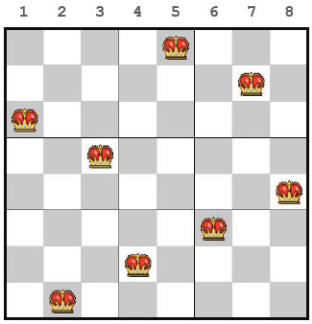
3-bit rotary encoder



8-bit rotary encoder

**Enumerating Permutations.**
**8-queens problem.** Place 8 queens on a chessboard so that no queen can attack any other queen.






Represent the solution as a permutation: q[i] = column of a queen in row i.
Queens **i** and **j** can attack each other if **|q[i] + i| = |q[j] + j|**
Enumerating Permutations. Given *n* items, enumerate all *n*! permutations (in each presentation, order matters)

*3-element permutations*

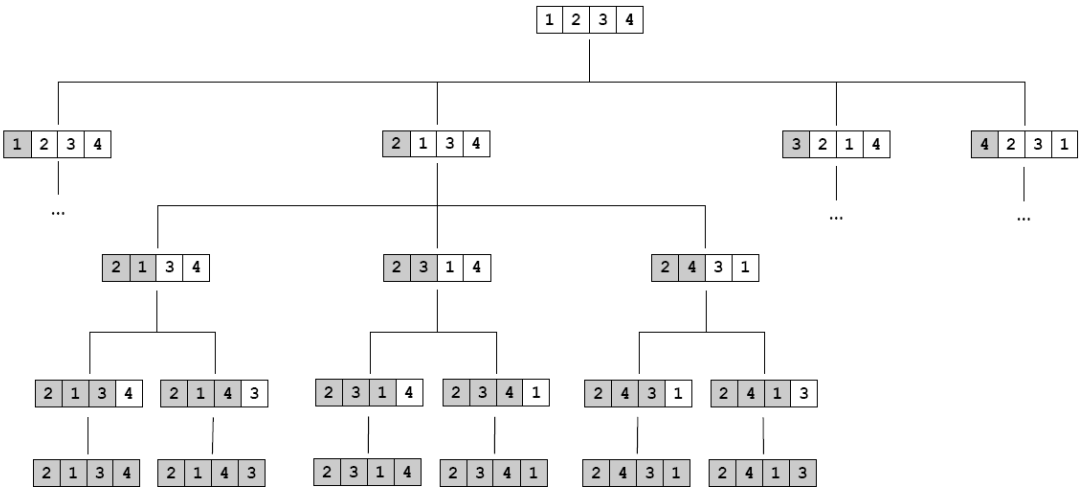
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 3 | 2 |
| 2 | 1 | 3 |
| 2 | 3 | 1 |
| 3 | 1 | 2 |
| 3 | 2 | 1 |

*4-element permutations*

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 2 | 1 | 3 | 4 | 3 | 1 | 2 | 4 | 3 | 1 | 2 | 4 |
| 1 | 2 | 4 | 3 | 2 | 1 | 4 | 3 | 3 | 1 | 4 | 2 | 3 | 1 | 4 | 2 |
| 1 | 3 | 2 | 4 | 2 | 3 | 1 | 4 | 3 | 2 | 1 | 4 | 3 | 2 | 1 | 4 |
| 1 | 3 | 4 | 2 | 2 | 3 | 4 | 1 | 3 | 2 | 4 | 1 | 3 | 2 | 4 | 1 |
| 1 | 4 | 2 | 3 | 2 | 4 | 1 | 3 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 |
| 1 | 4 | 3 | 2 | 2 | 4 | 3 | 1 | 3 | 4 | 2 | 1 | 3 | 4 | 2 | 1 |

1 followed by any permutation of 2 3 4 | 2 followed by any permutation of 1 3 4 | 3 followed by any permutation of 1 2 4 | 3 followed by any permutation of 1 2 4

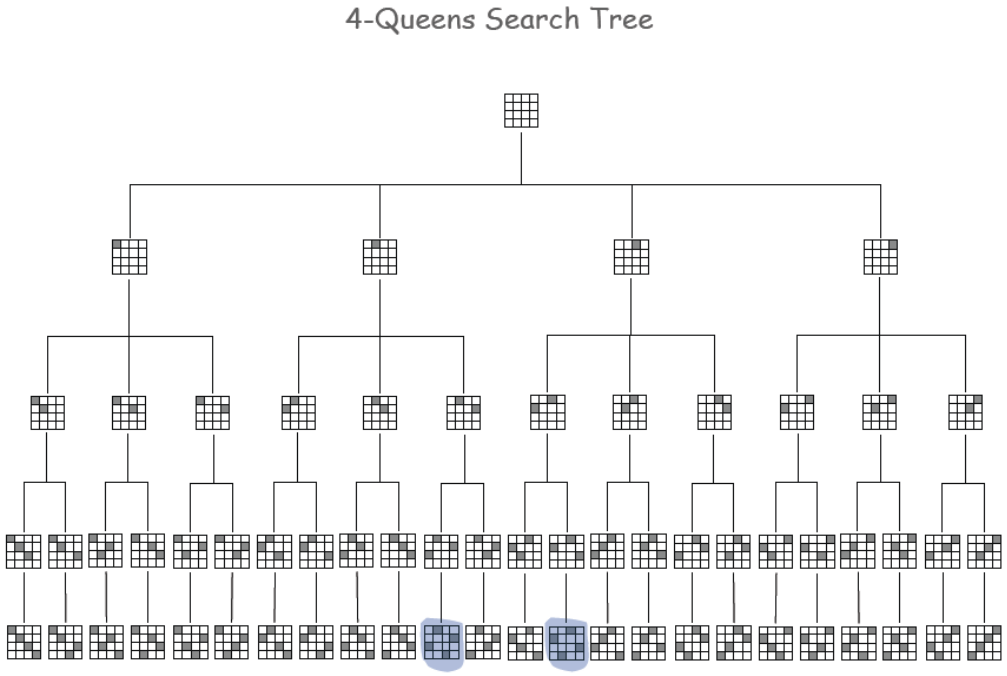
To enumerate all permutations of a set of *n* elements:
For each element $a_i$
– put $a_i$ first, then append
– a permutation of the remaining elements ($a_0$, …, $a_{i-1}$, $a_{i+1}$, …, $a_{n-1}$)

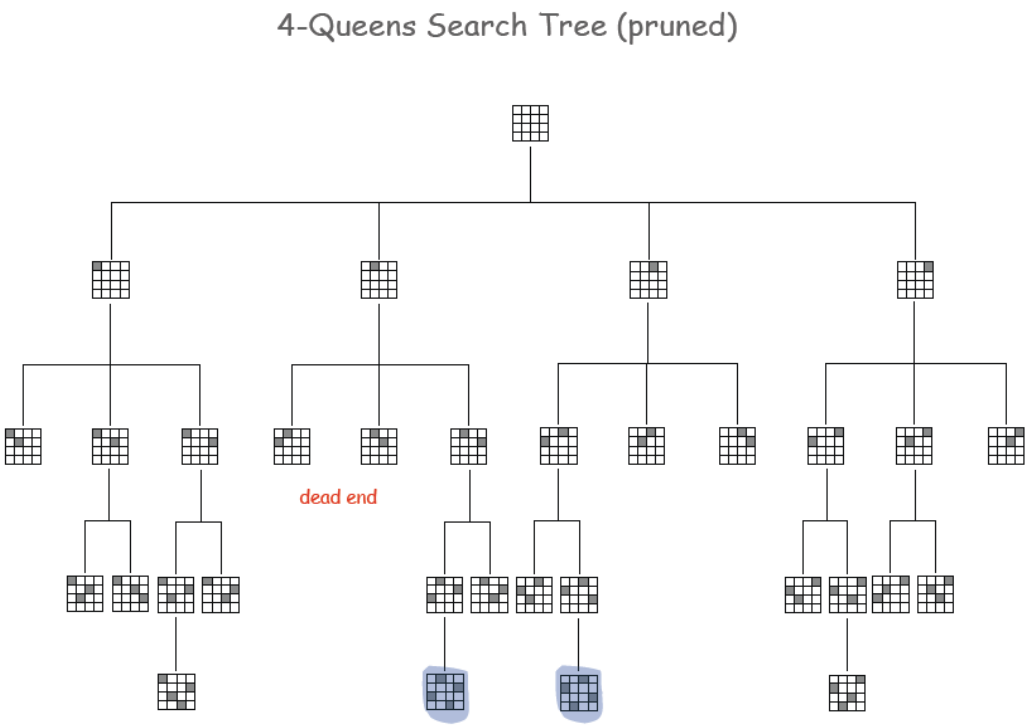


```
#include<iostream>
using namespace std;

int N = 4;
int a[] = {0, 1, 2, 3, 4};

void printPermutations(int a[])
{
 for(int i=1; i<=N; i++)
  cout << a[i];
 cout << endl;
}

void enumerate(int a[], int n)
{
 if (n == N) printPermutations(a);
 for (int i = n; i <= N; i++)
 {
  swap(a[i], a[n]);
  enumerate(a, n+1);
  swap(a[n], a[i]);
 }
}

int main()
{
 enumerate(a, 1);
}
```

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

**Pruning**






**N-Queens: Backtracking Solution**
Stop enumerating if adding the *n*-th queen leads to a violation.

```
void enumerate(int q[], int n)
{
  if (n == N) printQueens(q);
  for (int i = n; i <= N; i++)
   {
    swap(q[i], q[n]);
    if (isConsistent(q, n)) enumerate(q, n+1);
    swap(q[n], q[i]);
   }
}
```

**Sudoku:** Fill 9-by-9 grid so that every row, column, and box contains the digits 1 through 9.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 8 | | | | 3 | | |
| | | | 2 | | 1 | | | |
| 5 | | | | | | | | |
| | 4 | | | | | | 2 | 6 |
| 3 | | | | 8 | | | | |
| | | | 1 | | | | 9 | |
| | 9 | | 6 | | | | | 4 |
| | | | | 7 | | 5 | | |
| | | | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 2 | 8 | 9 | 4 | 6 | 3 | 1 | 5 |
| 9 | 3 | 4 | 2 | 5 | 1 | 6 | 7 | 8 |
| 5 | 1 | 6 | 7 | 3 | 8 | 2 | 4 | 9 |
| 1 | 4 | 7 | 5 | 9 | 3 | 8 | 2 | 6 |
| 3 | 6 | 9 | 4 | 8 | 2 | 1 | 5 | 7 |
| 8 | 5 | 2 | 1 | 6 | 7 | 4 | 9 | 3 |
| 2 | 9 | 3 | 6 | 1 | 5 | 7 | 8 | 4 |
| 4 | 8 | 1 | 3 | 7 | 9 | 5 | 6 | 2 |
| 6 | 7 | 5 | 8 | 2 | 4 | 9 | 3 | 1 |

**Solution:** Linearize. Treat 9-by-9 array as an array of length 81.


Enumerate all assignments. Count from 0 to $9^{81} - 1$ in base 9 (using digits 1 to 9).

Sudoku: Backtracking Solution
Iterate through elements of search space.

- For each empty cell, there are 9 possible choices.
- Make one choice and recur.
- If you reach a contradiction, go back to previous choice, and make next available choice.

Pruning. Stop as soon as you reach a contradiction.
Improvements: Choose most constrained cell to examine next.

**Hashing**

**Task:**
Given a constant $C$ and a sequence $a_1, a_2, .., a_n$, find if there exists a pair $a_i, a_j$, such that $a_i + a_j = C$.

**Hashing: Basic Plan**
Save items in a key-indexed table. Index is a function of the key.
**Hash function**. Method for computing table index from key.
**Collision resolution strategy**. Algorithm and data structure to handle two keys that hash to the same index.
**Classic space-time tradeoff.**

- No space limitation: trivial hash function with key as address.
- No time limitation: trivial collision resolution with sequential search.
- Limitations on both time and space: hashing (the real world).

**Choosing a Good Hash Function:**
Idealistic goal: scramble the keys uniformly.

- Efficiently computable.
- Each table position equally likely for each key.

Ex: Social Security numbers. Bad: first three digits. Better: last three digits.
Ex: date of birth. Bad: birth year. Better: birthday.
Ex: phone numbers. Bad: first three digits. Better: last three digits.

**Hash code.** A 32-bit int (between -2147483648 and 2147483647).
**Hash function.** An int between 0 and M-1.

```
String s = "call";
int code = s.hashCode();
int hash = code % M;
```

Bug. Don't use (`code % M`) as array index.
Subtle bug. Don't use (`abs(code) % M`) as array index.
OK. Safe to use (`(code & 0x7fffffff) % M`) as array index.

API for **hashCode()**.

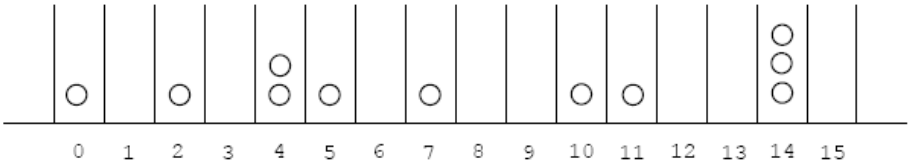
- Return an int.
- If x.equals(y) then x and y must have the same hash code.
- Repeated calls to x.hashCode() must return the same value.

**Default implementation.** Memory address of x.

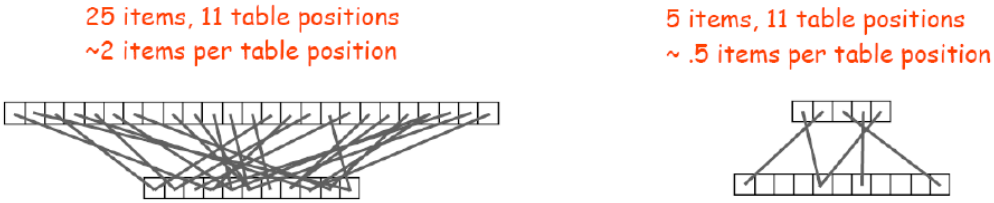
**Customized implementations. String, URL, Integer, Date.**

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
  hash = (31 * hash) + s[i];
return hash;
```

**Bins and balls. Throw balls uniformly at random into M bins.**

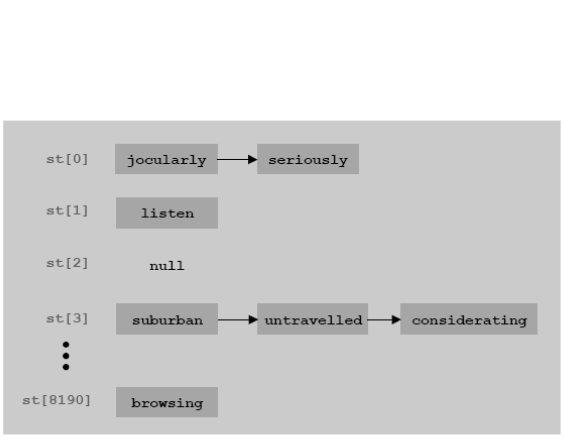



**Collision.** Two distinct keys hashing to same index**.**
**Conclusion.** Birthday problem – can't avoid collisions unless you have a ridiculous amount of memory.
**Challenge.** Deal with collisions efficiently**.**

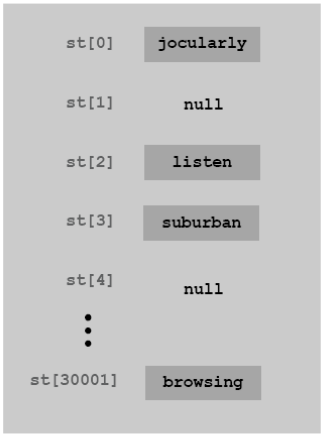



**Collision Resolution:** Two Approaches

**Separate chaining.** [H. P. Luhn, IBM 1953]
Put keys that collide in a list associated with index.
**Open addressing**. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]
When a new key collides, find next empty slot, and put it there.




separate chaining (M = 8191, N = 15000)




linear probing (M = 30001, N = 15000)

Separate chaining: array of M linked lists (typically M is approx. N/10)
Hash: map key to integer i between 0 and M–1.
Insert: put at front of i-th chain (if not already there).
Search: only need to search i-th chain.

**Separate chaining performance.**

- Cost is proportional to length of chain.
- Average length = N / M.
- Worst case: all keys hash to same chain.

**Theorem.** Let a = N / M > 1 be average length of list. For any t > 1, probability that list length > t a is exponentially small in t.

**Parameters.**

- M too large → too many empty chains.
- M too small → chains too long.
- Typical choice: a = N / M ≈ 10 → constant-time ops.

**Advantages.** Fast insertion, fast search.
**Disadvantage.** Hash table has fixed size, assumes good hash function.

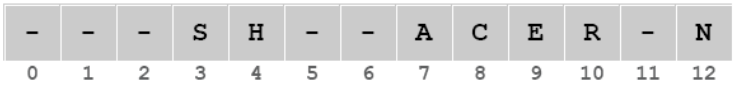
| | Worst Case | | | Average Case | | |
|---|---|---|---|---|---|---|
| Implementation | Get | Put | Remove | Get | Put | Remove |
| Sorted array | log N | N | N | log N | N/2 | N/2 |
| Unsorted list | N | N | N | N/2 | N | N/2 |
| Separate chaining | N | N | N | 1* | 1* | 1* |

* assumes hash function is random

**Linear probing: array of size M.**

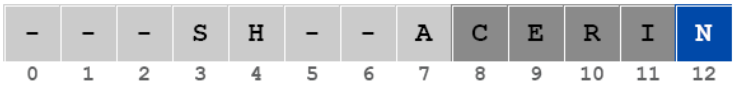
**Hash: map key to integer i between 0 and M-1.**

- Insert: put in slot i if free; if not try i+1, i+2, etc.
- Search: search slot i; if occupied but no match, try i+1, i+2, etc.



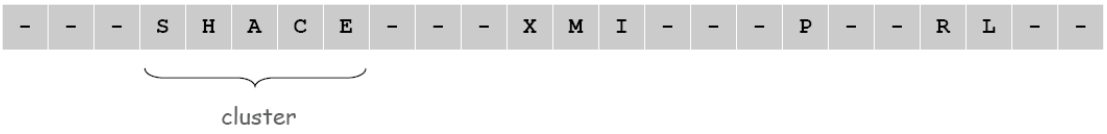


```
insert I
hash(I) = 11
```




```
insert N
hash(N) = 8
```

**Clustering**
**Cluster.** A contiguous block of items.
**Observation**. New keys likely to hash into middle of big clusters.




**Knuth's parking problem.** Cars arrive at one-way street with M parking spaces. Each desires a random space i: if space i is taken, try i+1, i+2, …

**What is mean displacement of a car?**

With M/2 cars, mean displacement is ≈ 3/2.
With M cars, mean displacement is ≈ (1/4) sqrt(2π M)

**Linear probing performance.**

- Insert and search cost depend on length of cluster.
- Average length of cluster =a = N / M.
- Worst case: all keys hash to same cluster.

**Parameters.**

- M too large → too many empty array entries.
- M too small → clusters coalesce.
- Typical choice: M ≈ 2N → constant-time ops

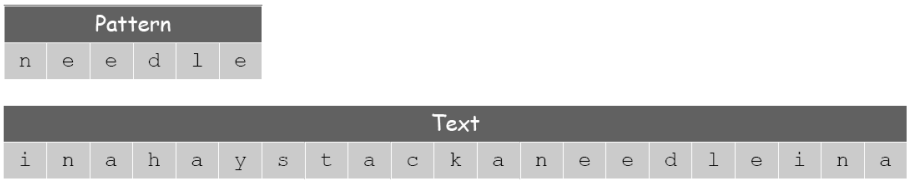
**Advantages.** Fast insertion, fast search.
**Disadvantage.** Hash table has fixed size, assumes good hash function.

| | Worst Case | | | Average Case | | |
|---|---|---|---|---|---|---|
| Implementation | Get | Put | Remove | Get | Put | Remove |
| Sorted array | log N | N | N | log N | N / 2 | N / 2 |
| Unsorted list | N | N | N | N / 2 | N | N / 2 |
| Separate chaining | N | N | N | 1* | 1* | 1* |
| Linear probing | N | N | N | 1* | 1* | 1* |

* assumes hash function is random

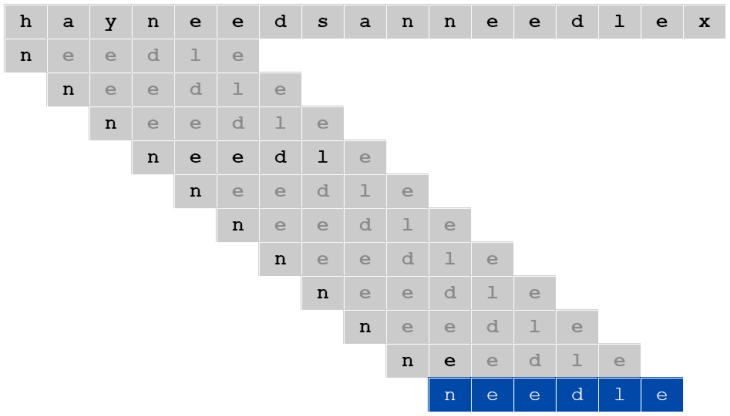
**String processing. String Search**

**String search**. Given a pattern string, find first match in text.
**Model.** Can't afford to preprocess the text.
**Parameters**. N = length of text, M = length of pattern.

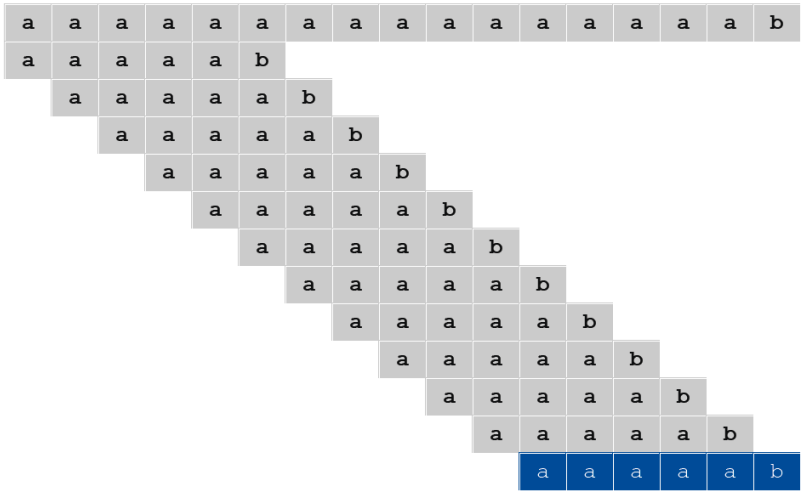



M = 6, N = 21

**Brute Force: Typical Case. Check for pattern starting at every text position.**




**Brute Force: Worst Case**




**Analysis of brute force.**

- Running time depends on pattern and text.
- Slow if M and N are large, and have lots of repetition.

character comparisons

| Implementation | Typical | Worst |
|---|---|---|
| Brute | 1.1 N † | M N |

Search for M-character pattern in N-character text

† assumes appropriate model

**Karp-Rabin Randomized Fingerprint Algorithm**
**Idea: use hashing.**

- Compute hash function for each text position.
- No explicit hash table: just compare with pattern hash!

Ex. Hash "table" size = 97

| Pattern | | | | |
|---|---|---|---|---|
| 5 | 9 | 2 | 6 | 5 |

59265 % 97 = 95

| Text | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | 7 | 9 | 3 | 2 | 3 | 8 | 4 | 6 |
| 3 | 1 | 4 | 1 | 5 | | | | | | | | | | | | | | | |
| | 1 | 4 | 1 | 5 | 9 | | | | | | | | | | | | | | |
| | | 4 | 1 | 5 | 9 | 2 | | | | | | | | | | | | | |
| | | | 1 | 5 | 9 | 2 | 6 | | | | | | | | | | | | |
| | | | | 5 | 9 | 2 | 6 | 5 | | | | | | | | | | | |

31415 % 97 = 84
14159 % 97 = 94
41592 % 97 = 76
15926 % 97 = 18
59265 % 97 = 95

**Brute force.** O(M) arithmetic ops per hash.
**Faster method to compute hash of adjacent substrings.**

- Use previous hash to compute next hash.
- O(1) time per hash.

Ex.
Pre-computed: **10000 % 97 = 9**
Previous hash: **41592 % 97 = 76**
Next hash: **15926 % 97 = ??**

Observation.

15926 % 97 = (41592 – (4 * 10000)) * 10 + 6
=(76 – (4 * 9 )) * 10 + 6
=406
=18

**False match.** Hash of pattern collides with another substring.
59265 % 97 = 95
59362 % 97 = 95

**How to choose modulus p.**
p too small → many false matches.
p too large → too much arithmetic.
Ex: p = 8355967 avoid 32-bit integer overflow.
Ex: p = 35888607147294757 avoid 64-bit integer overflow.

**Theorem.** If MN ≥ 29 and p is a random prime between 1 and $MN^2$, then Pr[false match] ≤ 2.53/N.

**Randomized algorithm. Choose table size p at random to be huge prime.**
**Monte Carlo. Don't bother checking for false matches.**

- Guaranteed to be fast: O(M + N).
- Expected to be correct (but false match possible).

**Las Vegas.** Upon hash match, do full compare; if false match, try again with new random prime.

- Guaranteed to be correct.
- Expected to be fast: O(M + N).

**Karp-Rabin summary.**

- Create fingerprint of each substring and compare fingerprints.
- Expected running time is linear.
- Idea generalizes, e.g., to 2D patterns.

character comparisons

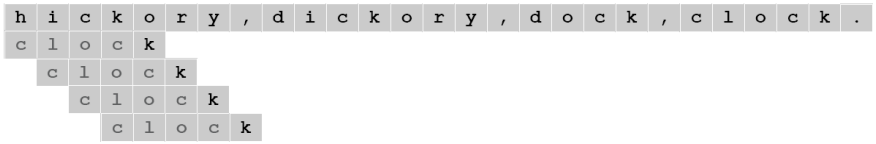
| Implementation | Typical | Worst |
|---|---|---|
| Brute | 1.1 N † | M N |
| Karp-Rabin | Θ(N) | Θ(N) ‡ |

Search for M-character pattern in N-character text
† assumes appropriate model
‡ randomized

**Boyer-Moore**

**Right-to-left scanning.**

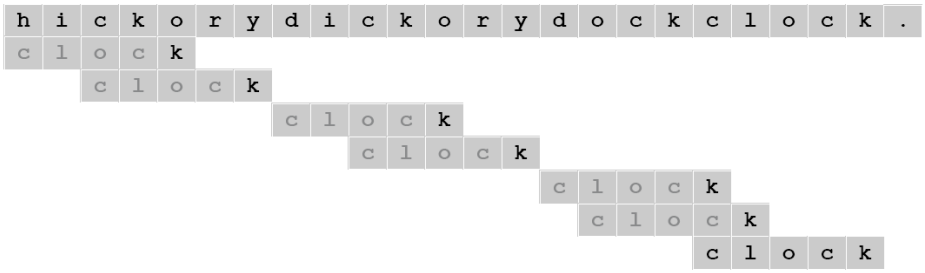
- Find offset i in text by moving left to right.
- Compare pattern to text by moving j right to left.




**Bad character rule.**

- Use right-to-left scanning.
- Upon mismatch of text character c, increase offset so that character c in pattern lines up with text character c.
- Precompute right[c] = rightmost occurrence of c in pattern.

| right[] | |
|---|---|
| c | 3 |
| k | 4 |
| l | 1 |
| o | 2 |
| * | -1 |




**Bad character rule analysis.**

- Highly effective in practice, particularly for English text: O(N / M).
- Takes O(MN) time in worst case.