Recitation 13 — Hashing and Dynamic Programming

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

39 November 2011

Today's Agenda:

- Hashing
- Dynamic programming

Announcements:

- Assignment 9 out this week. Last assignment of the semester!
  Due: Thursday, December 8.
- Final exam Monday, December 12, 5:30pm-8:30pm, WeH 7500

Questions from class?

1 Hashing

Let's review the high level ideas we covered before the break.

We have a large space of keys K (this might be infinite as in the case of the set of all possible strings, or simply large like all binary strings of length 1024) and a target range $\{0, \ldots, m-1\}$. We normally expect $|K| \gg m$. A hash function h is a mapping from K to $\{0, \ldots, m-1\}$.

One important application of hash functions is in implementing a dictionary data structure (hash table).

What are some properties we want out of a hash function?

- It should be a deterministic function—if we ask for the hash value of the same key twice, we should get the same value.
- It should distribute keys across buckets uniformly.

In general, there's a tension between wanting "random-like" behavior and wanting determinism. For now, we'll suppose we have a "good" hash function (without going too deep into what that means). For concreteness, though, we will make the following assumption:

> **The Simple Uniform Hashing Assumption.** Any given key $k \in K$ is equally likely to be hashed to one of the m possible values, independently of what values other keys might hash to.

Let's now address the other half of the problem, collision resolution.

Q: What does "load factor" refer to and why does it matter?

A: The ratio n/m. Literally, it is how full is the hash table is. It is also an indicator of how often we should expect a collision.

Q: What are some techniques for dealing with a collision?

A: Separate chaining and open addressing. In separate chaining, we build a so-called "chain"—a sequence or a list containing all the keys that hash to the same value; therefore, with this method, insertion, search, and deletion operations require traversing such a chain and has running time proportional to the length of the chain. In the other strategy, open addressing employs a technique that fits each key to a slot in a single array. For example, when a key k is inserted, we first consider the hashed-to location $h(k)$ and if that is occupied, we proceed to a different location according to some *probe sequence* until an empty slot is found. Searching is done in the same manner.

Q: Can you suggest some probe sequence patterns?

A: Formally, a probe sequence is a function $h(k, i)$ which decides the i-th alternative location for key k; thus, for a key k, we'd consider locations $h(k, 1), h(k, 2), \ldots$ in this order.

Linear probing is the simplest and most widely used probe sequence strategy, whereby $h(k, i) = [h(k) + (i - 1)] \bmod m$. That is, first it tries $h(k)$, then $h(k) + 1$, then $h(k) + 2$, so on so forth. This sequence wraps around if we reach the end of the array. This strategy has very good cache locality and performs really well in practice. In fact, it also has many nice theoretical performance guarantees, which we won't be able to cover here. Because of its simplicity, linear probing is often the method of choice in generic hash-table implementations these days.

Alternatively, we have *quadratic probing* where $h(k, i) = [h(k) + (i - 1)^2] \bmod m$, or *double hashing* where $h(k, i) = [h(k) + (i - 1) \cdot g(k)] \bmod m$ utilizing a second hash function g. In theory, all these techniques perform within constant factors of each other in expectation and are ideal in different sets of conditions.

Okay, great! It sounds like everyone remembers hashing basics. Let's move on to an application.

1.1 Example application: removing duplicates

Suppose we have a sequence of n elements possibly with some duplicate entries.

```
{ "quux", "foo", "bar", "foo", "baz", "bar" }
```

We want to remove the duplicates. Brute force: look at all pairs. How can we do better? Which pairs do we really need to look at?

Q: What's the relationship between two keys *having the same hash value* and *being equal*?

A: Being equal implies having the same hash index, but not the other way around.

Already, we have reduced our solution space by only comparing those keys that have the same hash.

Q: If we use separate chaining, how long do we expect a chain to be (at least intuitively)?

A: With the simple uniform hashing assumption, the probability that a key k hashes to a value t is $1/m$; therefore, the length of a chain is simply the number of keys, out of n keys, that hash to the same value t. We know by linearity of expectation that this is n/m.

This leads to the following algorithm: We're going to hash all the values in our sequence, keep all the elements in a bucket by themselves, and compare only those values within buckets.

Aside (ignore if you wish): Note that if the chains have lengths $B_1, B_2, \ldots, B_m$. Comparing entries within chain i will take $O(B_i^2)$ work; therefore, *even if* $\mathbf{E}[B_i] = n/m$ may be constant, the expected work $\mathbf{E}[B_i^2]$ can be much larger. Analyzing this requires looking at the "second moment" of B_i.

On our example above, let's suppose "bar" and "baz" hash to the same index. We'd get

1	2	3
foo, foo	bar, baz, bar	quux

Q: How can we implement this in parallel?

A: Hmm... separate chaining is non-trivial to implement in parallel.

Fortunately, there is a way use open addressing to give $O(n)$ work and logarithmic span.

Q: Recall, how did we hashed in parallel using open addressing?

A: The key idea was to put an element x into the hash table, only if the hash table was empty at $h(x, i)$. Every element attempts to write to the table in parallel. Anything that didn't get hashed in one round is hashed in the next round using the next probe position $h(x, i + 1)$. Rounds continued until every element was put in the hash table.

This approach used `injectCond` where the first element to write to an index "wins". There is no overwriting. (Recall that `inject` allows elements to overwrite other elements in the sequence; the last element to write at an a index "wins".)

Q: How could we use the same idea to check for duplicates?

A: Each element attempts to write its value in the hash table. Let's say the initial input consists of n elements and we're using a table of size m. As will be apparent, we'll want m to be larger than n to guarantee efficiency.

Q: What do we know about these values?

A: They must be unique, since only one of equal values can write to the hash table. Notice, though, that m is larger than n, so in this case, to extract out these unique elements, we're generally better off filtering on the input elements (cost: about n) instead of looking at the hashed-to table (cost: about m).

Q: How does an element know if it was the one that wrote to the hash table?

A: It writes not only its value, but also its index in the sequence. That is, element S_i writes (S_i, i) into location $h(S_i)$.

Q: How does an element know if it is a duplicate?

A: It has the same key as the element in the hash table but not the same index. That is, element S_j is a duplicate if what got written in $h(S_j)$ is (x, ℓ) where $x = S_j$ but $\ell \neq j$.

Q: Do all unique elements get written to the hash table?

A: No. Some may collide with elements in the hash table. This situation is similar to what happened in open addressing; however, we'll resolve the conflict differently.

Q: What can we do with elements that are not duplicates and have not been added to the hash table?

A: Repeat the process until there are no elements left.

In summary, using contraction, we proceed in rounds, where each round does the following:

1. For $i = 0, \ldots, |S| - 1$, each element S_i attempts to "write" the value (S_i, i) into location $h(S_i)$ in an array using `injectCond`.
2. We will divide S into unique elements (ACCEPT) and potentially unique elements (RETRY) as follows:

$$\mathsf{ACCEPT} = \langle S_i : H[h(S_i)] = (S_i, i)\rangle$$
$$\mathsf{RETRY} = \langle S_i : \#1(H[h(S_i)]) \neq S_i\rangle$$

3. Recurse on RETRY, appending together all the ACCEPT's.

The ACCEPT elements are those that successfully wrote to the hash table, and the RETRY elements are those that attempted to but did not write to the hash table and are not a duplicate of an element in ACCEPT. *It is crucial that* RETRY *does not contain a duplicate of an element in* ACCEPT. Further, note that implicitly, there is the other group REJECT which is thrown away: this group is made up of the elements that are duplicates of what we already have in ACCEPT.

Why is this algorithm correct? It is easy to see that if a key k is present in S, we'll never throw it away until we include it in ACCEPT, so we only need to argue that we never put two copies of the same key in ACCEPT. Let's consider a round of this algorithm. If S_i and S_j are the same

key, only one of them will be in ACCEPT, and furthermore, none of the entries of this key can be in RETRY.

We'll now analyze this algorithm: To bound work, we're interested in knowing how large RETRY is on an input sequence S of length n. Although the worst case might be bad, we're happy with expected-case behaviors. For this, it suffices to compute the probability that an entry S_i is included in RETRY. This happens *only if* the key S_i hashes to the same value as some other key S_j where $S_i \neq S_j$. Therefore, with the simple uniform hashing assumption, we have that for any i,

$$\begin{aligned}\mathbf{Pr}\left[S_i \in \mathsf{RETRY}\right] &\leq \mathbf{Pr}\left[\exists j \text{ s.t. } S_i \neq S_j \wedge h(S_i) = h(S_j)\right] \\ &\leq \sum_{j:S_j \neq S_i} \mathbf{Pr}\left[h(S_i) = h(S_j)\right] \\ &\leq n/m,\end{aligned}$$

where we have upperbounded the probability with a union bound.

If $m = 3n/2$, then $n/m = 2/3$, and by linearity of expectation, we have $|\mathsf{RETRY}| \leq 2n/3$. We have seen this recurrence pattern before; this gives that the total work is expected $O(n)$ because in expectation, this forms a geometrically decreasing sequence. Furthermore, applying KUW, we know that the number iterations is expected $O(\log n)$.

2 Dynamic Programming

Dynamic programming is a technique to avoid needless recomputation of answers to subproblems.

Q: When is it applicable?

A: When the computation DAG has a lot of sharing. Or when subproblems *overlap*.

So far, we haven't told you how to actually take advantage of overlapping solutions to get the efficiency gain. What we're doing right now is just steps 1 and 2 of DP, recognizing the inductive structure and the existence of the overlap. Let's review.

In class yesterday, we looked at the subset sum problem: given a set S and a number k, is there a subset of S whose elements sum to k?

Here's the code that produces a yes-or-no answer (remember, no actual DP yet):

```
(* SS : int list -> int -> bool *)

fun SS(S, k) =
  case (S, k) of
    (_, 0) => true
```

```
  | ([], _) => false
  | (s::S', k) =>
      if (s > k) then SS(S', k)
      else SS(S',k) orelse SS(S',k-s)
```

Let's tweak this a little to return the actual subsets rather than blind ourselves with booleans.

```
(* SS : int list -> int -> int list option *)

fun SS(S, k) =
  case (S, k) of
    (_, 0) => SOME []
  | ([], _) => NONE
  | (s::S', k) =>
      if (s > k) then SS(S', k)
      else case(SS(S',k), SS(S',k-s)) of
        (SOME L, _) => SOME L
      | (_, SOME L) => SOME (s::L)
      | _ => NONE
```

Another thing we might be interested in is how *many* solutions there are. This is just another minor tweak:

```
(* SS : int list -> int -> int *)

fun SS(S, k) =
  case (S, k) of
    (_, 0) => 1
  | ([], _) => 0
  | (s::S', k) =>
      if (s > k) then SS(S', k)
      else SS(S',k) + SS(S',k-s))
```

2.1 Example: Longest Palindromic Subsequence

Given a string s, we want to find the longest subsequence ss of s that is a palindrome (reads the same front and back). The letters don't have to be consecutive.

Example: `QRAECETCAURP` has the palindromes `RR`, `RAEDEAR`, `RACECAR`, `RAEDEAR`, and many more.

Q: How many palindromes could there be?

A: An exponential number. Ugg.

Q: How do we keep track of all of them?

A: We don't. Instead, we simplify the problem space, and find the *length* of the longest palindrome. Later we can consider reconstructing the palindrome or one of them.

Q: What's step one of coming up with a DP solution?

A: Recognize the inductive structure of the problem.

Q: What are the base cases of being a palindrome?

A: A 1- or 0-length string.

Q: How do you get bigger palindromes from smaller ones?

A: Add the same letter to both ends.

From the top-down approach, this translates into checking whether the outer letters are the same.

Q: If they are, how do we proceed?

A: Add 2 to the recursive call on the string between them.

Q: What if they're not – how can we proceed?

A: We can move our starting position or our ending position. Take the max of these two subcalls.

In code:

```
(* longest : char seq -> int *)
fun longest s =
  let fun longest' (i,j) =
    if (j-i <= 1) then j-i
    else if (nth s i = (nth s (j-1)) then 2 + longest'(i+1,j-1)
    else max(longest'(i+1,j), longest'(i,j-1))
  in
    longest' s (0,|s|)
  end
```

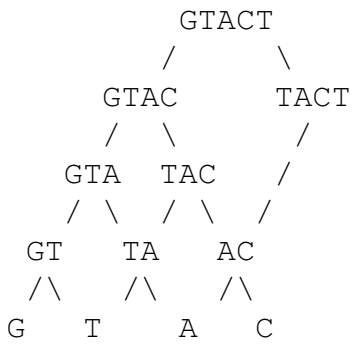
Intuitively, when we memoize, we'll want our table to be indexed by i and j so we can easily store and look up the longest palindromic subsequence between those two indices.

Q: What's the sharing structure here?

A: The two recursive calls share the **entire middle of the string**!

Let's look at an example:

```
AGTACTA
   |
```



With proper memoization, we only need to consider the number of vertices in the DAG of recursive calls and the work at each vertex to find the total work.

Q: How many vertices could there be in the worst case? That is, how many different arguments could there be to `longest'`?

A: $n * (n-1)/2$, since i and j can range between 0 and $n-1$, and $i \leq j$.

Since each call to `longest'` is constant work, the total work is $O(n^2)$.

Q: What is the span?

A: $O(n)$—since each time we invoke a subproblem, $j-i$ is at least one smaller and $0 \leq j-i \leq n-1$.

Exercise: how could we tweak this code to return the actual palindrome?