Recitation 14 – *Dynamic Programming! Hashing¡*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2015)

April 14th, 2015

1 Announce Dynamically!

- How was *SegmentLab* and the exam?
- *DPLab* just came out!

2 Dynamic Review!

Q: What is DP!

A: Dynamic programming is an algorithmic technique to avoid needless recomputation of answers to subproblems. DP problems have two identifying characteristics :

- *Inductive/recursive*. The solution to the larger problem instance is composed from solutions to smaller instances of the same problem.
- *Sharing*. The solution to each smaller problem instance can be used by multiple larger instances. This is what differentiates DP from other inductive techniques like divide-and-conquer. Avoiding needless recomputation means that we reduce the overall work of our algorithm.

How do we use these subproblems?

- We model our computation scheme as a **Directed Acyclic Graph**, a **DAG**, where each of our subproblems is a node, and we have a directed edge from problem A to B if the result of A depends on first computing B.
- Afterwards, we can solve a given problem by starting at the node representing our problem, S, and visiting all nodes in our DAG reachable from S. We call this the Top-Down approach.
- Alternatively, we can start from the bottom of the graph and work our way back up towards subproblems that depend on the current subproblem. We call this the Bottom-Up approach.
- Which method you use often does not make a difference except for space efficiency; however, there are times when one is better than the other due to data structure limitations, real-world implementations, or other limitations.

3 Dynamic Approach to a Dynamic Problem!

3.1 Problem : Longest Palindromic Subsequence (LPS)

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

Example: `QRAECDETCAURP` has inside it palindromes `RR`, `RADAR`, `RAEDEAR`, `RACECAR`, etc.

Q: How many palindromes could there be?

A: An exponential number.

Q: How do we keep track of all of them? (Trick question!)

A: We don't. Instead, we simplify the problem space, and find the *length* of the longest palindrome. Doing this will help us identify the inductive structure of the problem and sharing between subproblems. Later, we can consider recovering the longest palindrome (or if you are feeling adventurous, count unique ones).

3.2 Dynamic Solution!

In this section we describe a general approach to DP problems. As we go along, we'll also write up the solution according to the structure we'd like you to follow for assignments in this course (like *DPLab!*). Solutions should have 3 components:

1. Define a inductive/recursive solution to the problem, including base cases. Clearly state what each recursive call solves. Make sure that recursive calls are smaller than the function that is calling them. Often the recursive solution appears to be a brute-force solution.
2. Identify and argue that there is sharing among the parallel calls.
3. Analyze the DAG in terms of the number of nodes, the depth of the DAG, and the work and span of each node. The overall work is the sum of the work across the nodes, and the overall span is the sum of the spans along the longest path.

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

A: We want to be able to define our main problem instance in terms of smaller problem instances. This requires us to recognize the inductive structure of the problem.

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

A: A 1- or 0-length string, giving palindrones of length 1 and 0, respectively.

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

A: Add the same letter to both ends.

To generalize to finding a palindrone in a string S, we need to think about how to put subsolutions together. Often in DP it is helpful to consider solutions over contiguous subsequences or the input, and put these together.

Q: Why might contiguous subsequences be more helpful for the recursive structure than all subsequences?

A: The advantage is that although there are an exponential number of subsequences, there are only a polynomial number of contiguous subsequences. Therefore when we identify sharing, there are only a polynomial number of different contiguous subsequences to consider.

Q: How many contiguous subsequences are there for a sequence of length n

A: $n(n+1)/2$

Q: Returning to the recursive structure, what cases do we need to consider at the top level to find $LPS(S)$?

A: Either the first and last characters of S are equal, or they are not.

Q: What if they're equal – how can we proceed?

A: We can recursively solve the problem for the string between the two characters (a contiguous subsequences) and add 2 to account for the ends.

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

A: One of the letters may still be part of a palindromic subsequence. We therefore split into two branches, one for each end. In particular we consider the LPS by dropping the first character, and then by dropping the last. We can take the maximum of these.

Q: Argue that if the characters are the same, we do not have to consider the LPS by dropping the first or last character.

A: The longest palindromic subsequence found after dropping the last character, can be at most one longer than the one by dropping the first and last (since it only has one more character). However when including the 2 for matching the first and last, then including the 2 will always be better.

Q: Based on these observations, what is the recursive code?

As is often the case in DP, it is helpful to use a helper function, which uses integers to indicate the endpoints of a contiguous subsequence. We have:

```
fun LPS($S$) =
let
  % finds the longest palindromic subsequence for $S[i,\ldots,j]$
  %  i.e. the continguous subsequence of $S$ between locations $i$ and $j$, inclusive
  fun LPS'($i,j$) =
    if ($j < i$) then 0
    else if ($j = i$) then 1
    else if $S[i] = S[j]$ then $2 + \text{LPS}'(i+1, j-1)$
    else max(LPS'($i, j-1$), LPS'($i+1, j$))
    in  LPS(0, |S|−1)  end
```

A: Notice we just wrote down the four cases we mentioned above: two base cases, and the two inductive cases depending on whether the end characters are equal or not. Another syntax for $LPS'$with identical meaning which is often used in the specification of dynamic programs is:

$$\text{LPS'}(i,j) = \begin{cases} 0 & j < i \\ 1 & j = i \\ 2 + \text{LPS'}(i+1, j-1) & i < j \wedge S[i] = S[j] \\ \max(\text{LPS'}(i, j-1), \text{LPS'}(i+1, j)) & \text{otherwise} \end{cases}$$

Either syntax is acceptable for your written solutions on labs or exams, but note that the later is just for just LPS', and you still need to include everything else. In particular you must include the overall call, LPS(S), the comment describing the subproblem arguments (i.e. "finds the longest....") and the start case (i.e. LPS($0, |S|-1$)) in the order given.

Q: How much sharing is there?

A: For $|S| = n$ we only are going to call LPS' on $0 \le i < n, i-1 \le j < n$, i.e. on each contiguous subsequence. We have already determined there are only $O(n^2)$ of these.

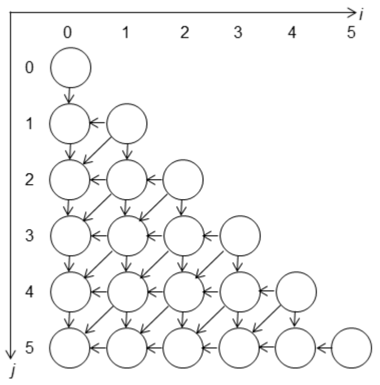
Q: What is the DAG structure and what are the costs?

A: There are $O(n^2)$ nodes in the DAG as determined above (one per subproblem instance), and the depth of the DAG is $O(n)$ since the recursion depth is at most n (each recursive call shrinks the string by at least 1). The cost of each node in the DAG is $O(1)$ for both work and span, since all the code is doing is a couple comparisons, a couple additions, and a maximum.

Therefore we have $W(n) = O(n^2)$, and $S(n) = O(n)$.

For illustrative purposes we can look at the DAG for `AGTACTA`:

If we place the vertices of the DAG into an matrix based on the their i and j the DAG will have the following structure, where each node can only depend on three neighbors (decrementing i, decrementing j, or decrementing both).



4 ¡More Dynamic Problems!

In this section, we tackle the classic Longest Increasing Subsequence (LIS) problem with our newfound DP chops. As usual, a subsequence of $A = (a_1, a_2, ..., a_n)$ is $(a_{i_1}, a_{i_2}, ..., a_{i_k})$ where $i_1 < i_2 < ... < i_k$. A subsequence is increasing if $A_{i_j} < A_{i_{j+1}}$ for $1 \leq j < k$.

Given the sequence A, we want to find the longest increasing subsequence (the one with maximum k). We write $n = |A|$ for simplicity.

More intuitively, we will cross out some numbers from A. We want the remaining numbers to be increasing. What is the fewest number of numbers we need to cross out?

One idea is a brute force solution: generate all possible subsequences, filter out the increasing ones, and select the longest one. This is prohibitively slow and painful to code; we can do better.

As in the longest palindromic subsequence problem we focus on contiguous subsequence. In particular we consider each prefix of a string, i.e. each contiguous subsequence $A[0,\ldots,j]$ for $j < n$, and consider a LIS whose last element is at $A[j]$. Call this $L(j)$.

Q: Suppose we knew $L(j)$ for $j < n$, how would we calculate $LIS(A)$?

A: $LIS(A) = \max_{i=0}^{n-1} L(i)$. Notice that we need the results for every i to compute the answer.

Q: What is the base case?

A: $L(0) = 1$ because the longest increasing subsequence ending with the first element contains just the element itself.

Q: Given $L(j)$ for all $j < i$, how can we compute $L(i)$?

A: $L(i) = 1 + \max_{0 \leq j < i, A[j] < A[i]} L(j)$. We quantify over all $L(j)$ where $A[j] < A[i]$ because we can only add $A[i]$ onto the subsequence ending at $A[j]$ if $A[i] > A[j]$. We take the max, and add one to indicate that $A[i]$ is added.

Q: What would the code for this look like?

A:

Q: How much sharing is there?

A: Well there are only $|A|$ possible values of i in $L(i)$ so there can be at most that many distinct function calls.

Q: What is the DAG structure and what are the costs?

A: The DAG has depth and size $n = |A|$. Therefore there is no parallelism in the DAG itself. However there is plenty of parallelism within each node. Notice that each node basically does a reduce over up to n values. Therefore the work of each node is $O(n)$ and the span is $O(\log n)$

Therefore overall we have $W(n) = O(n^2)$ and $S(n) = O(n \log n)$.

5 Hashing Review¡

We have a large space S_{keys} of keys, and a target range $\{0, \ldots, m-1\}$. We typically expect $|S_{keys}| \gg m$ (S_{keys} may even be infinite). A hash function is a mapping from S_{keys} to $\{0, \ldots, m-1\}$.

Q: What are the desired properties of a hash function?

A: It should be *deterministic*, and it should distribute keys uniformly across the target range.

Q: What is the "load factor" of a hash table, and what does it indicate?

A: The ratio n/m, where n is the number of keys that have been inserted, and m is the size of the table. This value indicates how full the hash table is, and consequently how often we should expect a collision.

Q: What are some possible techniques for handling collisions?

A: *Separate chaining* and *open addressing*.

Separate chaining forms lists ("chains") of keys that all map to the same hash value. Insertion, search, and deletion then all have running time proportional to the length of the chain.

Open addressing fits each key into one array slot. If a key k is not found at the initial location given by the hash function, then it might be stored at the next location in the *probe sequence*. The probe sequence is defined by a function, where $h(k,i)$ is the i^{th} alternative location for the key k.

Q: What are some examples of different probe sequences?

A: *Linear probing* is the simplest and most widely-used strategy, where $h(k,i) = [h(k)+i] \bmod m$. *Quadratic probing* defines $h(k,i) = [h(k)+i^2] \bmod m$. *Double hashing* utilizes a second hash function, g, and defines $h(k,i) = [h(k)+i \cdot g(k)] \bmod m$.

In theory, all of these techniques perform within constant factors of one another in expectation, and are ideal in different sets of conditions.

6 Parallel Hashing¡

Q: What do we mean by parallel hashing?

A: Instead of finding, inserting, or deleting one key at a time, each operation takes a set of keys and performs the operations on all of the keys in parallel. In this context, we have to be especially careful about collisions.

Q: How might we parallelize open addressing?

A: We perform multiple rounds. For `insert`, each round attempts to write the keys into the table at their appropriate positions in parallel. Any key that cannot be written because of a collision continues into the next round. We repeat until every key has been written into the table.

In order to prevent writing to a position already occupied in the table, we introduce a variant of the `inject` function. The function

$$\texttt{injectCond}(IV, S) : (\texttt{int} \times \alpha)\ \texttt{seq} \times (\alpha\ \texttt{option})\ \texttt{seq} \rightarrow (\alpha\ \texttt{option})\ \texttt{seq}$$

takes a sequence of index-value pairs $\langle (i_1, v_1), \ldots, (i_n, v_n) \rangle$ and a target sequence S and conditionally writes each value v_j into location i_j of S. In particular it writes the value only if the location is set to NONE and there is no previous equal index in IV. That is, it conditionally writes the value for the *first* occurrence of an index; recall `inject` uses the *last* occurrence of an index.

For example, if $S = \langle \texttt{SOME}\ 5, \texttt{NONE}, \texttt{NONE}, \texttt{SOME}\ 42, \texttt{NONE}, \texttt{SOME}\ 28 \rangle$, then
$\texttt{injectCond}(\langle (3,43), (1,97), (4,8), (1,35) \rangle, S) = \langle \texttt{SOME}\ 5, \texttt{SOME}\ 97, \texttt{NONE}, \texttt{SOME}\ 42, \texttt{SOME}\ 8, \texttt{SOME}\ 28 \rangle$.

Let's write `insert`. Let T be our hash table and K be the set of keys we wish to insert.

```
fun insert($T$,$K$) =
let
  fun insert'($T$,$K$,$i$) =
    if $|K| = 0$ then $T$
    else let
      val $T'$ = injectCond($\{(h(k,i),k) : k \in K\}$, $T$)
      val $K'$ = $\{k : k \in K \mid T[h(k,i)] \neq k\}$
    in
      insert'($T'$,$K'$,$i+1$)
    end
in
  insert'($T$,$K$,0)
end
```

For round i (starting at $i = 0$), `insert` attempts to put each key k into the hash table at position $h(k,i)$, but only if the position is empty. To see whether it successfully wrote a key to the table, it reads the values written to the table and checks if they are the same as the keys. In this way it can filter out all the keys that it successfully wrote to the table. It repeats the process on any keys that did not get hashed on the next round using their next probe position $h(k, i+1)$. Rounds continue until every element was put in the hash table.

For example, suppose the table has the following entries before round i:

$T =$

0	1	2	3	4	5	6	7
	A		B			D	F

If $K = \langle E, C \rangle$ and $h(E,i)$ is 1 and $h(C,i)$ is 2, then $IV = \langle (1,E), (2,C) \rangle$ and `insert'` would fail to write E to index 1 but would succeed in writing C to index 2, resulting in the following table:

$T' =$

0	1	2	3	4	5	6	7
	A	C	B			D	F

It then repeats the process with $K' = \langle E \rangle$ and $i + 1$.

Note that if T is implemented using an `stseq`, then parallel `insert` basically does the same work as the sequential version which adds the keys one by one. The difference is that the parallel version may add keys to the table in a different order than the sequential. For example, with linear probing, the parallel version adds C first using 1 probe and then adds E at index 4 using 4 probes:

$T_P =$

0	1	2	3	4	5	6	7
	A	C	B	E		D	F

Whereas, the sequential version might add E first using 2 probes, and then C using 3 probes:

	0	1	2	3	4	5	6	7
$T_S =$		A	E	B	C		D	F

Both make 5 probes in the table. Since we showed that, with suitable hash functions and load factors, the expected cost of insert is $O(1)$, the expected work for the parallel version is $O(|K|)$. In addition, if the hash table is large enough, the expected size of K decreases by a constant fraction in each round, so the span is $O(\log |K|)$.

7 Example Hashing¡ Application: Removing Duplicates

In class, one of the examples that showed good speedup was removing duplicates. That is, suppose we have a sequence of n elements possibly with some duplicate entries, and we want to remove the duplicates.

$$\langle \text{“}quux\text{”}, \text{“}foo\text{”}, \text{“}bar\text{”}, \text{“}foo\text{”}, \text{“}baz\text{”}, \text{“}bar\text{”} \rangle$$
$$\Downarrow$$
$$\langle \text{“}quux\text{”}, \text{“}foo\text{”}, \text{“}bar\text{”}, \text{“}baz\text{”} \rangle$$

A brute force solution would look at all pairs, but this is clearly inefficient. 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 $\Rightarrow$ having the same hash index, but having the same hash index $\nRightarrow$ being equal.

Already, we have reduced our solution space by only comparing those keys that have the same hash. This leads to the following algorithm: hash all the values in our sequence and compare only those that fall within the same bucket. We'll use open addressing, and will be able to achieve $O(n)$ work and $O(\log n)$ span!

Q: How might we use parallel open addressing to check for duplicates?

A: Each element attempts to write its value in the hash table. If two keys are equal, only one of them gets successfully hashed.

Q: How do we proceed to the next round in order to eliminate duplicates?

A: When collecting the keys to retry inserting in the next round (by comparing each element to the one at its hashed position in the array), exclude duplicates of elements that already got hashed.

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

A: Instead of just writing elements into the table, we write a pair (v, i) where v is the value and i is its index in the original sequence. If two elements are duplicates, only one of them will get written with its index. The other has the same key as the element written to 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, so we repeat the process until there are no elements left. However, unlike hashing with open addressing, we can start the second round with an empty hash table after collecting the elements that successfully hashed, since these are definitely in our non-duplicate sequence.

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:

$$\begin{aligned}\mathsf{ACCEPT} &= \langle S_i \in S \mid T[h(S_i)] = (S_i, i)\rangle \\ \mathsf{RETRY} &= \langle S_i \in S \mid \#1(T[h(S_i)]) \neq S_i\rangle\end{aligned}$$

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, if we assume simple uniform hashing, we have that for any i,

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

where we have upper bounded 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, we can show that the number iterations is $O(\log n)$ in expectation.

8 Bonus¡!

How could you implement parallelism for the top-down approach? Memoization in parallel gets tricky. How will you handle fetching unknown results? What type of system level locks would you have to use to synchronize that fetching? Do you notice any inefficiencies in the naive approach? If you're interested, look up *futures and promises*.

Operations on hashing-based tables and balanced-tree-based tables generally have a logarithmic factor difference in cost. What capabilities do we lose? What happens if you were to try and add these capabilities back in? Can you think of how these properties might be useful or necessary?