Algorithms by Jeff E: Dynamic Programming

Siddharth Bhat

Monsoon, second year of the plague

Question: 2a: number of partitions into words.

```
// substr_co = substring in closed-open interval.
int q2a(string s) {
  const int n = s.size();
  vector<int> dp(n+1, 0);
  // dp[ls] = number of words in s[0:ls).
  // ls for ``length of s''
  for(int ls = 1; ls <= n; ++ls) {
      for(int j = 0; j < ls; ++j) { // break off word [j..ls)
          if (!isword(s.substr_co(j, ls)) { continue; }
          nwords[ls] = max(nwords[ls], 1 + nwords[j-1]);
      }
  }
  return nwords[n];
}
```

Question: 2b: decide if s, t can be partitioned into words at same indexes.

```
// substr_co = substring in closed-open interval.
int q2b(string s, string t) {
  const int n = s.size();
  vector<int> dp(n+1, 0);
  // dp[i] = number of words in s[0:i)
  for(int l = 1; l <= n; ++l) {
      for(int j = 0; j < l; ++j) { // break off word [j, l)
          if (!isword(s.substr_co(j, l)])  ||
              !isword(t.substr_co(j, l)])) { continue; }
          nwords[l] = max(nwords[l], 1 + nwords[j-1]);
      }
  }
  return nwords[n] > 0;
}
```

Question: 2c: number of ways to partition s, t into words at same indexes.

```
// cc = closed-closed interval.
int q2c(string s, string t) {
  const int n = s.size();
  vector<int> dp(n+1, 0);
  // dp[i] = number of words in s[0:i)
  for(int i = 1; i <= n+1 ++i) {
      for(int j = 1; j <= i - 1; ++j) {
          if (!isword(s[cc(j, i-1)])  || !isword(t[cc(j, i-1)])) { continue; }
          nwords[i] = max(nwords[i], 1 + nwords[j-1]);
      }
  }
  return nwords[n];
}
```

Question: 3a: largest sum subarray. Standard solution: kendane's algorithm

```
int q3a(const vector<int> &xs) {
    const int n = xs.size();
    int best = 0;
    int cur = 0;
    for(int i = 0; i < n; ++i) {
        if (cur + xs[i] < 0) { continue; }
        cur += xs[i];
        best = max<int>(best, cur);
    }
    return best;
}
```

$\Theta(n^2)$ solution using

Question: 3b: largest product subarray.

I have a somewhat dubious $\Theta(n)$ solution. I'd love to know a correctness proof.

```
int best = 1, pos = 1, neg = 1;
int q3b(const vector<int> &xs) {
    const int n = xs.size();
    for(int i = 0; i < xs.size(); ++i) {
      if (xs[i] == 0) { pos = neg = 1; }
      else if (xs[i] < 0) {
           const int prevpos = pos;
           pos = neg  * xs[i]; neg = prevpos * xs[i];
       } else if (xs[i] > 0) { pos *= xs[i]; neg *= xs[i]; }
      best = max<int>(best, pos);
    }
    return best;
}
```

Question: 4a: largest sum circular subarray. Use kendane's algorithm/sliding window on `xs`$\diamond$`xs` with a window length restriction of $|xs|$.

```
int q3a(const vector<int> &xs) {
    xs.append(xs);

    const int n = xs.size();
    int best = 0;
    int cur = 0;
    left = 0;
    for(int i = 0; i < 2*n; ++i) {
        if (cur + xs[i] < 0) { left = i; continue; }
        cur += xs[i];
        // closed-closed: [left, cur]
        // maintain length of |n|
        if (cur - left + 1 > n)  {
            cur -= xs[left]; left++;
        }

        best = max<int>(best, cur);
    }
    return best;
}
```

Question: 4b: long subarray sum. Use kendane's algorithm/sliding window on `xs` with a window length restriction that it must be greater than X.

```
int q3a(const vector<int> &xs, int X) {
    xs.append(xs);

    const int n = xs.size();
    int best = 0;
    int cur = 0;
    left = 0;
    for(int i = 0; i < 2*n; ++i) {
        if (cur + xs[i] < 0) { left = i; continue; }
        cur += xs[i];
        // closed-closed: [left, cur]
        // only consider if has #elem > X
        if (cur - left + 1 >= X)  {
          best = max<int>(best, cur);
       }
   }
    return best;
}
```

Question: 4c: short subarray sum. Use kendane's algorithm/sliding window on `xs` with a window length restriction that it must be at most X.

```
int q3a(const vector<int> &xs, int X) {
    const int n = xs.size();
    int best = 0;
    int cur = 0;
    left = 0;
    for(int i = 0; i < 2*n; ++i) {
        if (cur + xs[i] < 0) { left = i; continue; }
        cur += xs[i];
        // closed-closed: [left, cur]
        // only consider if has #elem <= X
        if (cur - left + 1 <= X)  {
            best = max<int>(best, cur);
        }
    }
    return best;
}
```

Question: 4d: small subarray sum. This is much more challenging, since the subarray sum is not a monotonic quantity. The best I have is a naive $\Theta(n^2)$ algorithm that tries all subarrays.

```
int q3a(const vector<int> &xs, int X) {
    const int n = xs.size();
    int best = 0;
    for(int i = 0; i < n; ++i) {
        int sum = 0;
        for(int j = i; j < n; ++j) {
            sum += xs[j];
            if (sum < X) { best = max<int>(best, sum); }
        }
    }
    return best;
}
```

Question: 5a: longest common subsequence. Standard DP.

```
int q5a(const vector<int> &xs, const vector<int> &ys) {
    // dp[lx][ly]: LCS between xs[0:lx) and ys[0:ly) [closed-open].
    // lx, ly for ``length of xs, length of ys''
    vector<vector<int>> dp(1+xs.size(), vector<int>(1+ys.size(), 0));
    int best = 0;
    for(int lx = 1; lx <= xs.size(); ++lx) {
        for(int ly = 1; ly <= ys.size(); ++ly) {
            dp[lx][ly] = max(dp[lx-1][ly-1], dp[lx][ly-1], dp[lx-1][ly]);
            if (xs[lx-1] == ys[ly-1]) {
                dp[lx][ly] = max(dp[lx][ly], 1 + dp[lx-1][ly-1]);
            }
        }
    }
    return dp[xs.size()][ys.size()];
}
```

Question: 5b: longest common supersequence. Standard DP.

```
int q(const vector<int> &xs, const vector<int> &ys) {
    // dp[lx][ly]:
    // longest common super-sequence
    // between xs[0:lx) and ys[0:ly) [closed-open].
    // lx, ly for ``length of xs, length of ys''
    const int INFTY = 1e9;
    vector<vector<int>> dp(1+xs.size(), vector<int>(1+ys.size(), INFTY));
    int best = 0;
    for(int lx = 1; lx <= xs.size(); ++lx) {
        for(int ly = 1; ly <= ys.size(); ++ly) {
            if (xs[lx-1] == ys[ly-1]) {
                // add one letter (the equal letter).
                dp[lx][ly] = min(dp[lx][ly], 1 + dp[lx-1][ly-1]);
            } else {
                // add two letters (x and y).
                dp[lx][ly] = min(2 + dp[lx-1][ly-1], dp[lx][ly]);
            }
            // try adding only x / only y, punting other letter.
            dp[lx][ly] = min(1 + dp[lx][ly-1], dp[lx][ly]);
            dp[lx][ly] = min(1 + dp[lx-1][ly], dp[lx][ly]);

        }
    }
    return dp[xs.size()][ys.size()];
}
```

Question: 5c: longest bitonic subsequence. Use naive $\Theta(n^2)$ LIS to compute longest increasing subsequence in $x[0:i]$ and longest decreasing subsequence at $x[i:N]$. Combine these to get longest bitonic sequence at i as $\texttt{lis}(i)+\texttt{lds}(i)-1$. We need a -1 to prevent over-counting of the middle i term.

```
// out[i]: lis of xs[0:i]
int mklis(vector<int> &xs, vector<int> &out) {
    const int n = xs.size();
    out.reserve(n, 0);
    out[0] = 1;
    for(int i = 1; i < n; ++l) {
         // loop invariant: out[i] contains length of LIS
         // of subsequence from xs[0:i).
         out[i] = out[i-1]; // can make at least as long.
         for(int j = 0; j < i; ++j) {
            if (xs[i] > xs[j]) { // use xs[i].
               out[i] = max(out[i], out[j]+1);
            }
         }
    }
}
// similarly write mklds which computes lds[i:xs.size())
int q(const vector<int> &xs) {
    vector<int> liss; mklis(xs, liss);
    vector<int> ldss; mklds(xs, ldss);
    int best = 1;
    for(int i = 0; i < xs.size() ++i) {
      best = max(best, liss[i] + ldss[i] - 1);
    }
    return best;
}
```

Question: 5d: longest oscillating subsequence. For even i: $xs[i] < xs[i+1]$. For odd i, $xs[i] > xs[i+1]$.

```
int q(const vector<int> &xs) {
    const int n = xs.size();
    // dp[i][0]: length of LOS of subseq of xs[i:n]
    //           with first of LOS having even index.
    // dp[i][1]: length of LOS of subseq of xs[i:n]
    //           with first term of LOS having odd index.
    vector<pair<int, int>> dp(n, 0);
    dp[n-1][0] = dp[n-1][1] = 1; // can always use highest index.
    for(int i = n-2; i >= 0; i--) {
        for(int j = i+1; j < n; ++j) {
          dp[i][0] max (dp[i][0], dp[j][0]); // no new elem
          dp[i][1] max (dp[i][1], dp[j][1]); // no new elem.
          if (xs[i] < xs[j]) {
              // we are in the even index. j will be odd index.
              dp[i][0] = max(dp[i][0], dp[j][1] + 1);
          } else if (xs[i] > xs[j]) {
              // we are in the odd index. j will be even index.
              dp[i][1] = max(dp[i][1], dp[j][0] + 1);
          }
        }
    }
    int best = 0;
    for(int i = 0; i < n; ++i) {
        best = max(best, dp[i][0], dp[i][1]);
    }
    return best;
}
```

Question: 5d: shortest oscillating super-sequence. For even i: $xs[i] < xs[i+1]$. For odd i, $xs[i] > xs[i+1]$. TODO TODO!

Question: 5e: longest convex subsequence. Find a subsequence X such that $2X[i] < X[i-1]+X[i+1]$. I propose a $\Theta(n^3)$ solution. Surely faster solutions exist?

```
int conv(int ip1, int i, const vector<int> &xs) {
    assert(i < ip1);
    // is always possible to have any sequence that just uses [i, ip1].
    int best = 2;
    for(int im1 = 0; im1 < i; ++im1) {
        if (2*xs[i] < xs[im1] + xs[ip1]) {
            best = max(best, 1 + conv(i, im1));
        }
    }
    return best;
}

int f(vector<int> &xs) {
  const int n = xs.size();
  if (n <= 2) { return n; }
  // dp the above conv() recurrence.
  int best = 0;
  for(int ip1 = 0; ip1 < n; ++i) {
    for(int i = 0; i < ip1; ++i) {
      best = max<int>(best, conv(ip1, i));
    }
  }
  return best;
}
```