The University of Michigan
Electrical Engineering & Computer Science
EECS 281: Data Structures and Algorithms
Spring 2024

Lab 10: Algorithm Families and Dynamic Programming

Instructions:
Work on this document with your group, then enter the answers on the canvas quiz.

Note:
This lab assignment contains a survey, multiple choice quiz, and coding portion. Submit your answers to the Lab 10 Canvas quiz and the coding portion to the autograder. You will have **three** attempts on the quiz.

You MUST include the following assignment identifier at the top of every file you submit to the autograder as a comment. This includes all source files, header files, and your Makefile (if there is one). If there is no autograder assignment, you may ignore this.

Project Identifier: D7E20F91029D0CB08715A2C54A782E0E8DF829BF

---

1 Logistics

1. What date is the final exam?
    A. December 10, 2024
    B. December 13, 2024
    C. December 15, 2024
    D. December 2, 2035
2. What date is the lab 10 autograder due?
    A. November 25, 2024
    B. November 30, 2024
    C. December 2, 2024
    D. December 1, 2024

2 Algorithm Families and Dynamic Programming

3. Which algorithm seeks to break problems into smaller, non-overlapping subproblems then recursively combine the answers to those problems for the solution?

    A. Greedy

    B. Branch and Bound

    C. Divide and Conquer

    D. Dynamic Programming

    E. Brute Force

4. Which algorithm exhaustively tries every possible solution to a problem in order to find the optimal solution?

    A. Greedy

    B. Branch and Bound

    C. Divide and Conquer

    D. Dynamic Programming

    E. Brute Force

5. Which algorithm will break a problem into subproblems and combine the solutions to those while recording the solutions to those subproblems in order to avoid solving the same subproblem twice?

    A. Greedy

    B. Branch and Bound

    C. Divide and Conquer

    D. Dynamic Programming

    E. Brute Force

6. Which algorithm will take a locally optimal choice at each step of the algorithm, but might not necessarily be globally optimal?

    A. Greedy

    B. Branch and Bound

    C. Divide and Conquer

    D. Dynamic Programming

    E. Brute Force

7. Which algorithm will discard potential solutions which are worse than the current best solution?

    A. Greedy

    B. Branch and Bound

    C. Divide and Conquer

    D. Dynamic Programming

    E. Brute Force

8. What kind of algorithms are Prim's and Kruskal's?

    A. Brute Force

    B. Greedy

    C. Divide and Conquer

    D. Branch and Bound

    E. None of the above

9. What kind of algorithm is selection sort?

    A. Brute Force

    B. Greedy

    C. Divide and Conquer

    D. Branch and Bound

    E. None of the above

10. What kind of algorithms are quicksort and mergesort?

    A. Brute Force

    B. Greedy

    C. Divide and Conquer

    D. Branch and Bound

    E. None of the above

11. Your company is working on a map application but has run into a roadblock in their algorithm — finding the shortest route from point A to point B takes too long for practical use! Upon further analysis, you realize that the algorithm often does unnecessary work by traversing paths that are worse than the current optimal path. What algorithm can you use to fix this problem?

    A. Brute Force

    B. Greedy

    C. Divide and Conquer

    D. Branch and Bound

    E. None of the above

12. Which of the following statements is/are TRUE? Select all that apply.

    A. A brute force solution will always give you the optimal solution.

    B. Because backtracking avoids looking at large portions of the search space by pruning, the asymptotic complexity of backtracking is always better than that of brute force.

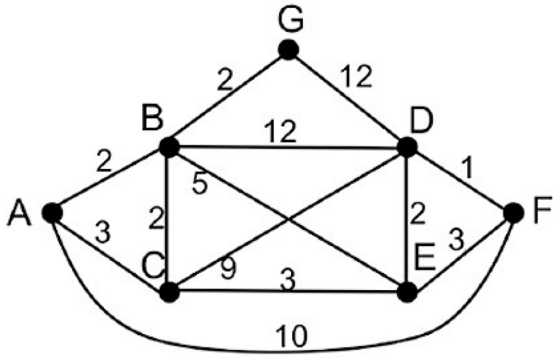
    C. The greedy algorithm guarantees an optimal solution to the 0-1 knapsack problem.

    D. Branch and bound will not speed up your program if it takes just as long to determine the bounds than to test all possible choices.

    E. Dynamic programming will always reduce both the time and memory needed to solve a problem with multiple overlapping subproblems.

    F. given n items and a knapsack capacity of m, the dynamic programming solution to the 0-1 knapsack problem runs in $\Theta(mn)$ time.

13. You are using Dijkstra's algorithm to find the shortest path from vertex A to every other vertex in the graph above. S is the set of vertices for which the minimum path weight from A has been found. A is the first vertex you add to S. Which of the following gives the correct order in which vertices are added to S?



    A. A, B, C, E, G, D, F

    B. A, B, G, C, E, D, F

    C. A, C, B, G, E, D, F

    D. A, B, G, D, C, E, F

    E. A, B, C, G, E, D, F

14. Consider the recurrence relation

$$T(n) = T(n-1) + T(n-3) + T(n-4)$$

    $[n > 4]$ with $T(n) = n$ for $1 <= n <= 4$. What is the time complexity of the best algorithm for calculating $T(n)$ with $O(n)$ additional space?

    A. $O(1)$

    B. $O(\log(n))$

    C. $O(n)$

    D. $O(n^2)$

    E. $O(3^n)$

15. Which of the following is **true** about dynamic programming?

    A. A dynamic programming solution for calculating the N^{th} fibonacci number can be implemented with $O(1)$ additional memory

    B. Dynamic programming is mainly useful for problems with disjoint subproblems

    C. A bottom-up DP solution to a problem will **always** use the same amount of stack space as a top-down solution to the same problem

    D. A top-down DP solution to a problem will **always** calculate every single subproblem.

3 Gradescope Problem: 0-1 Knapsack

This problem is to be submitted independently. We recommend trying it on your own, checking your answer with your group and discussing solutions, and then submitting it to Gradescope. These will be graded on completion, not by correctness. However, we want to see that you were thinking about the problem. Please implement your solution in ***knapsack.cpp****. The starter files can be found on Canvas.*

To practice DP, we would like you to try to come up with the DP solution to the 0-1 Knapsack Problem, as described in lecture. Don't just copy it from the slides, try to build it up yourself!

You have a bag of weight capacity (`cap`) and a selection of N items, which each have a value and a weight. Values and weights are passed in as vectors, where the i^{th} item has value of `value[i]` and weight of `weight[i]`.

Choose which items to put in your bag so that you maximize the combined value of all the items in your bag without going over the weight capacity. Return this maximum value.

Implementation: Your solution should run in $O(nC)$ time.

```
int knapsack(   int C,
                const std::vector<int> &value,
                const std::vector<int> &weight);
```


4 Coding Assignment: Meal or No Meal? A Trip to the ChEECScake Factory!

Your favorite restaurant, the ChEECScake Factory, has a customer loyalty program. On your first visit to the restaurant, you are offered a punch card. Whenever you purchase a meal at full price, you can add one hole punch to your punch card. Once you have five punches, you can turn in the card for a free meal and a brand new punch card. You will ONLY receive a brand new punch card when you turn in your old card for a free meal.

SUPER HELPFUL: If your function returns the value -281 you'll get HUGE help from the autograder. Work on this problem some **before** reading that help, but make sure that you get the help information BEFORE the day that the lab is due!! Getting the help counts as a submission.

For example, if your meals cost [3, 3, 3, 3, 3, 3, 3, 120], then you should earn hole punches on the first five meals ($15), pay normally for the next two meals ($6), and then turn in the punch card so that the $120 meal is free! The lowest cost would be $21 ($15 + $6).

However, you ALSO have a lot of coupons that can be used at the restaurant. In fact, you have enough coupons that you can apply one to every meal! If you apply a coupon, you get a 25% discount on that meal (rounded down to the nearest whole number). But there's a catch: if you apply a coupon, you don't get to add a hole punch to your punch card! Using the example above, you would be able to use coupons on the two $3 meals before the $120 meal, which would bring the lowest cost down to $19 ($15 + 0.75 * $3 + 0.75 * $3, rounded down).

Consider another example: if your meals cost [2, 2, 2, 2, 1000, 100] and you use the first five meals to earn hole punches, you would need to spend $1008 to get the $100 meal for free. It would be much better to apply the 25% discount to each item so that you pay a total of $829. Note that we apply the discount separately to each item and round each meal down to a whole number, so we'll pay $1 + $1 + $1 + $1 + $750 + $75 = $829.

There are, however, many cases where it would make sense to use a mixture of punch discounts and discounting coupons. This is where your program comes in!

Write a program that, given a vector of meal prices, calculates the MINIMUM cost needed to pay for all meals using both the loyalty program and restaurant coupons.

Implementing the Program

Download the following three starter files from the lab10 folder on Canvas:

- `deals.h`
- `Makefile`
- `samples.cpp`

You will be doing your coding in the `deals.h` file. Do not modify the other two files!

Here are some additional notes you should read before you begin coding:

- write your implementation in the provided `best_price()` function — the function should return the best price that you calculated

- do **NOT** print output in the `deals.h` file — remember to remove all cout statements before submitting
- use the provided `discounted()` function to compute the discount from applying a coupon
- you *must* either use a coupon or apply the punch card for each meal
- you have an unlimited number of coupons
- your program should run in linear time
- greedy solutions will not work (so don't attempt writing one) — instead, use dynamic programming!

Testing

Once you are done with your program, you may use the provided `Makefile` and `samples.cpp` file to test your implementation. The `samples.cpp` file includes 15 public test cases that you may use to check the accuracy of your program. Simply make an executable with `deals.h` and `samples.cpp` in the same directory and run it (using the command `./meals`). If done correctly, you will get output in the following format:

```
[meal  #1] ... PASS
[meal  #2] ... PASS
[meal  #3] ... PASS
[meal  #4] ... PASS
[meal  #5] ... PASS
[meal  #6] ... PASS
[meal  #7] ... PASS
[meal  #8] ... PASS
[meal  #9] ... FAIL: expected 33929 but got 33932
[meal #10] ... PASS
[meal #11] ... PASS
[meal #12] ... FAIL: expected 23422 but got 23426
[meal #13] ... FAIL: expected 9106 but got 9124
[meal #14] ... FAIL: expected 15306 but got 15308
[meal #15] ... PASS
```

This is just an example of what you might see; if done correctly, you will PASS all of the test cases! You should use this output to help you debug your code. To further help you with the debugging process, the comments under each test case in the `samples.cpp` file also provide the total cost with no discounts or promotions, the total cost with only coupon discounts, and the total cost if the punch card is always applied.

Submitting to the Autograder

Before submitting, add you and your partner's names to the top of the deals.h file. To submit to the autograder, create a `.tar.gz` file containing just deals.h:
`tar -czvf lab10.tar.gz deals.h`

If you are working with a partner, both partners must submit to the autograder. Only students who submit code to the autograder will receive points. It's perfectly fine for both partners to submit identical code, as long as the code was written by both of the partners.

You will be able to make three submissions to the autograder per day, up until the due date. If you are stuck, it may be helpful to submit a partial implementation early on...

there are some resources available that may require you to make a submission to view — this is much easier for you if you have many submissions left before the due date! Please do not wait until the last day to begin this — you can almost certainly expect a dynamic programming written question on the final exam, so an understanding of how to approach this problem (and similar problems like it) will be very important.

Hints and Advice

The Main Idea:

Dynamic programming may be a challenging concept at first, since it is very different from many other algorithms you've used before. The idea of DP is to determine if a smaller subproblem can be used to solve a larger problem. If this is possible, you can "remember" the solution to the smaller subproblem in order to solve the larger one.

Fibonacci Example:

Recall the Fibonacci numbers example from lecture:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$$

Each Fibonacci number is calculated by summing up the two Fibonacci numbers directly before it. For instance, the 10th number in the sequence, 34, is the sum of the 8^{th} and 9^{th} numbers in the sequence $(13 + 21 = 34)$. If we knew the 8^{th} and 9^{th} Fibonacci numbers, we would be able to quickly calculate the 10^{th} Fibonacci number. How can we calculate the 9^{th} Fibonacci number? Well, the 9^{th} number is the sum of the 7^{th} and 8^{th} Fibonacci numbers. If we knew the 7^{th} and 8^{th} Fibonacci numbers, we would be able to quickly calculate the 9^{th} Fibonacci number. How can we calculate the 8^{th} Fibonacci number? Well, the 8^{th} number is the sum of the 6^{th} and 7^{th} Fibonacci numbers. If we knew the 6^{th} and 7^{th} Fibonacci numbers, we would be able to quickly calculate the 8^{th} Fibonacci number.

Do you see where this is going? Eventually, we get to a value we can determine in constant time - a base case that we know right off the bat. In the case of the Fibonacci numbers, we know (by definition) that the first two Fibonacci numbers are 0 and 1. We can use this information to our advantage. For example, we can calculate the 3rd Fibonacci number by summing 0 and 1. Although we could be done here, it is important to note that future Fibonacci numbers depend on the value of the 3rd Fibonacci number. Thus, it would be to our advantage to store the value of the 3rd Fibonacci number somewhere (e.g. an array) so that we can refer to it quickly without having to calculate it again.

Applying DP to Lab 10:

The same idea applies to this problem. When dealing with dynamic programming, think small and try to build upwards. Suppose you want to calculate the cheapest cost for 1000 meals. This might seem like a daunting task at first, but do you know how to calculate the cheapest cost for 1 meal? You can do this in constant time. It turns out that you can use the solution for 1 meal to quickly calculate the cheapest cost for 2 meals. The cheapest cost for 2 meals can then be used to calculate the cheapest cost for 3 meals, and so on. If you store the best cost for n meals (in a memo), you can quickly retrieve that information to calculate the best cost for n+1 meals. Eventually, you'll build upwards to a point where you'll be able to calculate the cheapest cost for 1000 meals, using the information you've encountered before.

Note that this is different from the Fibonacci example because you have multiple stamps in play as well; the cheapest price to get to n meals and ending with 3 stamps on your punch card is different from the cheapest price to get to the same n meals and ending with 4 stamps on your punch card. Thus, for each meal, you also have to store the cheapest

cost associated with that meal under the condition that you end up with 0 stamps, with 1 stamp, with 2 stamps, and so on, on your punch card.

If you are still stuck, you can take the starter file we give you and return -281 in the `best_price` function (without making any other edits to the file). If you do, you will get some further tips from the autograder. However, doing this will use up a submission! It is also recommended that you attempt the problem before trying this, since you will not have help at your fingertips when we ask you a dynamic programming question on the exam!

One Last Note:
This is the last lab question you'll have to solve in EECS 281. Thank you for a fantastic semester, and we wish you all the best in your future endeavors! Please take the time to evaluate our performance after the end of the semester and complete this short survey about your experiences in lab and what we can do to improve for future semesters. Your feedback is greatly appreciated! Good luck on the final exam!

