9 Algorithm design techniques

9.1 Divide and conquer

Reading: MAW 10.2

- Divide: $P \Rightarrow P_1, \ldots, P_k$

  Conquer: $S(P_1), \ldots, S(P_k)$

  Merge: $S(P_1), \ldots, S(P_k) \Rightarrow S(P)$

- Examples: Sorting (merge sort and quick sort), searching (binary search), closest pair (the $O(n \log n)$ algorithm), and selection (the linear-time algorithm).

- Algorithm template:

```
function P(n)
  if n <=c
     solve P directly
     return  its solution
  else P => P1, ..., Pk   //divide
       for i = 1 to k
         Si = P(ni)       //conquer
       S1, ..., Sk => S   //merge
       return S
```

- Time complexity:

$$T(n) = \begin{cases} 1 & n \leq c \\ \sum_{i=1}^{k} T(n_i) + D(n) + M(n) & n > c \end{cases}$$

- Strassen's algorithm

  - Given $A = (a_{ij})_{n \times n}$ and $B = (b_{ij})_{n \times n}$.
    Let $C = A \times B = (c_{ij})_{n \times n}$, for $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.
  - First algorithm:

```
for i = 1 to n
  for j = 1 to n
    c[i,j] = 0
    for k = 1 to n
      c[i,j] = c[i,j] + a[i,k] * b[k,j]
```

    Time complexity: $O(n^3)$.
  - Second algorithm:

$$A_{n \times n} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$B_{n \times n} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C_{n \times n} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

So the multiplication of two $n \times n$ matrices becomes eight multiplications of two $\frac{n}{2} \times \frac{n}{2}$ matrices, giving us $T(n) = 8T(\frac{n}{2}) + O(n^2)$. By iterating, we have $T(n) = O(n^3)$. No improvement!

– Third (Strassen's) algorithm:

M_1	$(A_{12} - A_{22})(B_{21} + B_{22})$
M_2	$(A_{11} + A_{22})(B_{11} + B_{22})$
M_3	$(A_{11} - A_{21})(B_{11} + B_{12})$
M_4	$(A_{11} + A_{12})B_{22}$
M_5	$A_{11}(B_{12} - B_{22})$
M_6	$A_{22}(B_{21} - B_{11})$
M_7	$(A_{21} + A_{22})B_{11}$

C_{11}	$M_1 + M_2 - M_4 + M_6$
C_{12}	$M_4 + M_5$
C_{21}	$M_6 + M_7$
C_{22}	$M_2 - M_3 + M_5 - M_7$

Using the above idea in the algorithm, we get $T(n) = 7T(\frac{n}{2}) + O(n^2)$, thus $T(n) = O(n^{\log 7}) = O(n^{2.81})$ by iterating.

9.2 Greedy method

Reading: MAW 10.1

- Making change: How can a cashier make change in the amount of x to a customer using the smallest possible number of coins, given the coinage $\{1, 5, 10, 25\}$ and an unlimited supply of coins of each denomination?

  A greedy algorithm: Starting with nothing, at every stage we add to the coins already chosen a coin of the largest value available that does not take us past the amount to be paid.

  If $x = 94$, then the change includes 3 quarters, 1 dime, and 1 nickel and 4 pennies.

- Greedy algorithms are suitable for solving optimization problems with solutions consisting of a sequence of choices.

  Greedy algorithms make the choice that looks the best at the moment (short sighted), and never changes the choices made (stubborn). So they may not always give optimal solutions, yet they are usually simple to understand and implement.

- Two simple examples:

  – The knapsack problem: Given $U = \{u_1, \ldots, u_n\}$, where for each item $u_i \in U$, w_i is its weight and v_i is its value. Let W be the weight limit of the knapsack. Determine $U' \subseteq U$ such that $\sum_{U'} w_i \leq W$ and $\sum_{U'} v_i$ is maximized.

    Greedy algorithm 1: Always pick the item with the largest value.

    Greedy algorithm 2: Always pick the item with the largest value/weight.

  – What is the quickest way to go from A to B during rush hour in Manhattan?

    Greedy algorithms can be bad since it is sometimes quicker to make a temporary detour in the opposite direction.

- A job scheduling problem

  – Given jobs $J_1, \ldots, J_n$, where the length of J_i is t_i. How can we schedule the jobs on a machine such that the sum of the completion times (total completion time) of the jobs is minimized?

  – Consider the following example.

J_1	J_2	J_3	J_4	J_5
1	3	2	5	4

Schedule 1: J_1, J_2, J_3, J_4, J_5.

Total completion time $\sum c_i = 1+4+6+11+15 = 37$.

Schedule 2: J_1, J_3, J_2, J_5, J_4.

Total completion time $\sum c_i = 1+3+6+10+15 = 35$.

Observation: Each permutation of the n jobs is a schedule.

– A greedy algorithm: Shortest Job First (SJF).

  * Sort the job by increasing t_i.
  * Schedule the jobs on the machine in the sorted order.

  Time complexity: $O(n \log n)$.

  Is the algorithm optimal? (Does the algorithm always produce the schedule with the minimum total completion time for any input?)

– Proof of optimality:

  By contradiction, assume that the optimal schedule (OPT) is not the same as the greedy schedule (SJF). Assume $J_{\pi(1)}, \ldots, J_{\pi(n)}$ is the order in OPT. Then in OPT, there must be $J_{\pi(i)}$ and $J_{\pi(j)}$ such that $i < j$ but $t_{\pi(i)} > t_{\pi(i)}$. Now we swap these two jobs to get a new schedule NEW. Consider the difference of the total completion times of OPT and NEW,

$$\begin{aligned} \sum c_i(OPT) \quad &- \quad \sum c_i(NEW) \\ &= \quad (\# + 1)(t_{\pi(i)} - t_{\pi(j)}) \\ &> \quad 0, \end{aligned}$$

  where # is the number of jobs between $J_{\pi(i)}$ and $J_{\pi(j)}$ in OPT. We conclude that NEW is a better schedule than OPT. A contradiction to that OPT is optimal. So OPT should be the same as SJF. So SJF is optimal.

- Bin packing

  – Problem definition:

    Given $S = (s_1, \ldots, s_n)$, where $s_i \in (0,1]$ for $i = 1, \ldots, n$. Pack the numbers in the minimum number of unit-capacity bins.

    Let $S = (0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8)$. The optimal packing uses only three bins:

$$[0.2, 0.8], [0.7, 0.3], [0.4, 0.1, 0.5]$$

  – On-line bin packing: s_i must be placed into a bin before $s_{i+1}, \ldots, s_n$ are processed. In other words, when making decision about s_i, an on-line algorithm does not know anything about $s_{i+1}, \ldots, s_n$.

  – First-fit (FF):

```
for i = 1 to n
   C(Bi) = 0  // content in Bi
for i = 1 to n
   j = min {k: C(Bk)+si <=1} //*
   put si into Bj
   C(Bj) = C(Bj) + si
```

    The algorithm is greedy since a new bin is used only when no old bin can take the number.

    For $S = (0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8)$, FF uses four bins. So the algorithm is not an optimal algorithm, it is just a heuristic.

    What is the time complexity of FF?

  – Best-fit (BF):

    In FF, replace * with the following statement.

```
   choose j such that C(Bj)+si<=1 and
     C(Bh)+si<=C(Bj)+si for h = 1,...,n
```

For $S = (0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8)$, BF uses four bins. Therefore, BF is not optimal either.

What is the time complexity of BF?

– Next-fit (NF):

```
for i = 1 to n
   C(Bi) = 0
put s1 into B1
C(B1) = C(B1) + s1
k = 1
for i = 2 to n
   if C(Bk) + si <= 1
     put si into Bk
     C(Bk) = C(Bk) + si
   else k = k + 1
        put si into Bk
        C(Bk) = C(Bk) + si
```

For $S = (0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8)$, NF uses five bins. So NF is not optimal either. It is even worse than FF and BF for the input.

What is the time complexity of NF?

9.3 Dynamic Programming

Reading: MAW 10.3.1

- Divide-and-conquer algorithms are implemented by recursion. Its design is top-down, and it is efficient when the subproblems don't overlap. However, when subproblems do overlap (share sub-subproblems), recursion does redundant work. In this case, a tabular method is often used. It is nonrecursive and bottom-up. It is called dynamic programming.

- An example: Fibonacci numbers:

  $fib1(n)$

  if $n < 2$ return n

  else return $fib1(n-1) + fib1(n-2)$

  We can see that this recursive algorithm is not efficient. To compute $fib1(n)$, the algorithm computes $fib1(n-1)$ and $fib1(n-2)$ separately. To compute $fib1(n-1)$, the values of $fib1(n-2)$ and $fib1(n-3)$ are needed. To compute $fib1(n-2)$, the values of $fib1(n-3)$ and $fib1(n-4)$ are needed. We observe that subproblems $fib1(n-1)$ and $fib(n-2)$ share sub-subproblems.

  The time complexity $T(n) \geq T(n-1) + T(n-2)$. So $T(n)$ is larger than the nth Fibonacci number. So $T(n) \geq \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (-\frac{1+\sqrt{5}}{2})^{-n}) = O(1.618^n)$.

  We can use the dynamic programming method by building a 1-D table as below and returning the nth entry of the table.

k	0	1	2	3	4	$\cdots$	n
f_k	0	1	1	2	3	$\cdots$	f_n

```
fib2(n)
 if n < 2 return n
 else i = 0
      j = 1
      for k = 2 to n
        f = i+j
        i = j
        j = f
      return f
```

The time complexity is obviously $O(n)$.

- To summarize, dynamic programming follows the next five steps.

  1. Define a function F in words so that the solution information is embedded in $F(n)$.
  2. Define F recursively:
     $$F(n) = G(F(n_1), F(n_2), \ldots, F(n_k))$$
     for $n_1, n_2, \ldots, n_k < n$.
  3. Construct a table to compute nonrecursively $F(n_1), F(n_2), \ldots, F(n_k)$, hence $F(n)$.
  4. Give the algorithm in pseudocode.
  5. Analyze the algorithm complexity.

- Chained matrix multiplication

  – We wish to compute $A_1 \times A_2 \times \cdots \times A_n$, where A_i is a $p_{i-1} \times p_i$ matrix. Which order of computation should we use to achieve the highest efficiency of the algorithm?

  – The number of basic operations needed to compute $A_i \times A_{i+1}$ is $p_{i-1} p_i p_{i+1}$.

  – Order of computation determines the time efficiency. For example, $A_1 : 10 \times 20$, $A_2 : 20 \times 50$, $A_3 : 50 \times 1$, and $A_4 : 1 \times 100$. If we use the order in $A_1 \times (A_2 \times (A_3 \times A_4)$, the number of basic operations is $(50 \times 1 \times 100) + (20 \times 50 \times 100) + (10 \times 20 \times 100) = 125,000$. However, if we use the order in $((A_1 \times A_2) \times A_3) \times A_4$, the number of basic operations is $(10 \times 20 \times 50) + (10 \times 50 \times 1) + (10 \times 1 \times 100) = 11,500$.

  – Question: What is the minimum number of basic operations in computing $A_1 \times A_2 \times \cdots \times A_n$?

  – Let $m[i, j]$ be the minimum number of basic operations in computing $A_i \times A_{i+1} \times \cdots \times A_j$ for $1 \le i \le j \le n$. Assume in general that k is used to indicate the position of the last multiplication to be performed among all: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$. Then
    $m[i, j] = 0$ if $i = j$.
    $m[i, j] = \min_{i \le k \le j-1}\{m[i,k] + m[k+1, j]$
    $+ p_{i-1} p_k p_j\}$ if $i \ne j$.

  – We can use a dynamic programming algorithm to compute $m[1, n]$, the minimum number of basic operations in computing $A_1 \times A_2 \times \cdots \times A_n$. Entries are filled left to right and bottom to top. Note that those in the lower left triangle are undefined.

$i \backslash j$	1	2	$\cdots$	n
1	0	$\uparrow$	$\cdots$	$*$
2	$--$	0	$\cdots$	$\uparrow$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
n	$--$	$--$	$\cdots$	0

  – The algorithm:

```
for i = 1 to n
  m[i,i] = 0
for j = 2 to n
   for i = j-1 to 1
      m[i,j] = +infty
      for k = i to j-1
         temp = m[i,k] + m[k+1,j]
                       + p_{i-1}p_kp_j
         if temp < m[i,j]
           m[i,j] = temp
return m[1,n]
```

  – Time complexity: $O(n^3)$.

- The world series/NBA finals
  - Two teams are in the final to play the best of $2n-1$ format. The winner will be the first to achieve n victories. No draw is allowed. Assume that the results of each match are independent. Let p be the probability that A wins a game and $q = 1-p$ be the probability that B wins a game. What is the probability that A wins the series?
  - Let $P(i,j)$ be the probability that A wins the series given that A still needs i more wins to achieve the victory and B needs j more wins if B is to win the series. What $P(n,n)$?

    $P(i,0) = 0$ if $1 \leq i \leq n$.

    $P(0,j) = 1$ if $1 \leq j \leq n$.

    $P(i,j) = pP(i-1,j) + qP(i,j-1)$ otherwise.

    A table of size $(n+1) \times (n+1)$ can be constructed to store the values of $P(i,j)$.

```
for i = 1 to n
   P[i,0] = 0
for j = 1 to n
   P[0,j] = 1
for i = 1 to n
   for j = 1 to n
      P[i,j] = p * P[i-1,j] +
               q * P[i,j-1]
return P[n,n]
```