CS 5114
Final Exam Solutions

Meghendra Singh (meghs@vt.edu)

Given: May 5, 2017, 11:00AM **Due:** May 8, 2017, 1:00PM

Instructions

A. You must neither give not receive aid on this exam; to do otherwise is a violation of the Honor Code.

B. The exam consists of three problems, worth a total of 250 points.

C. Prepare your solutions electronically so as to produce a single PDF file.

D. You may consult the textbook, your notes, or the handouts.

E. Write your answers carefully. **In particular, use complete sentences.**

F. Upload your PDF solutions to Canvas by the due time.

Good luck!

[50] 1. Give asymptotic upper and lower bounds for $T(n)$ in the following recurrence:

$$T(n) \quad = \quad 4T(n/2) + n^2 3^n.$$

Make your bounds as tight as possible and prove them.

We can draw the recursion tree for the recurrence as shown in Figure 1 below:

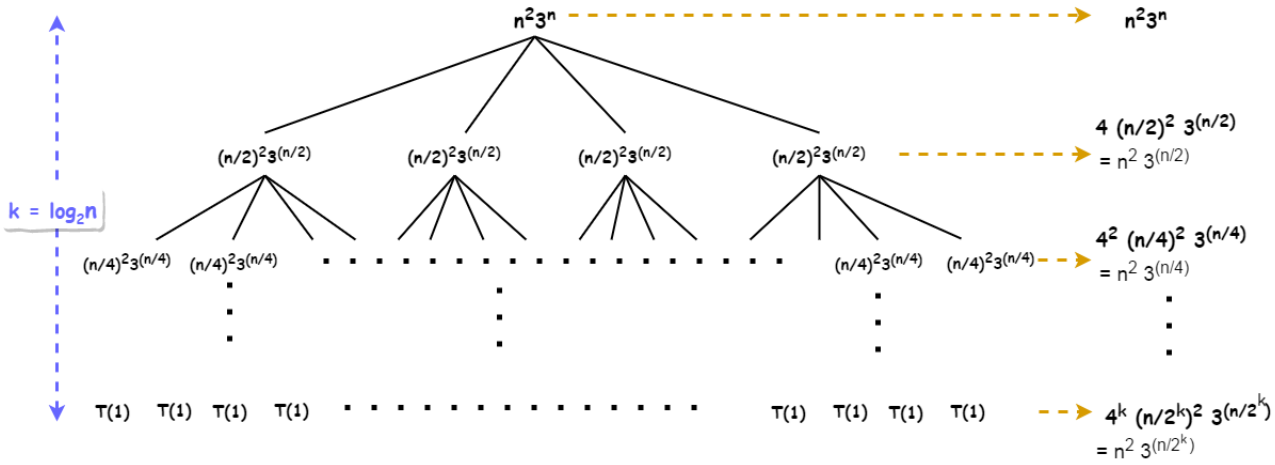

Figure 1: Recursion tree for $T(n) = 4T(n/2) + n^2 3^n$

We can write $T(n)$ as the sum of the costs till level k of the recursion tree (costs are shown in the rightmost part of figure 1), we get:

$$\begin{aligned} T(n) &= n^2 3^n + n^2 3^{\frac{n}{2}} + n^2 3^{\frac{n}{2^2}} + ... + n^2 3^{\frac{n}{2^{k-1}}} + n^2 3^{\frac{n}{2^k}} \\ &= n^2 (3^{\frac{n}{2^0}} + 3^{\frac{n}{2^1}} + 3^{\frac{n}{2^2}} + ... + 3^{\frac{n}{2^{k-1}}} + 3^{\frac{n}{2^k}}) \end{aligned}$$

$$T(n) = n^2 (3^{\frac{n}{2^k}} + 3^{\frac{n}{2^{k-1}}} + ... + 3^{\frac{n}{2^2}} + 3^{\frac{n}{2^1}} + 3^{\frac{n}{2^0}}) \tag{1}$$

Assuming that $T(1) = 1$ and the leaves of the recursion tree are at level k (as shown in figure 1), we can compute the value of k as follows:

$$\frac{n}{2^k} = 1$$

$$\implies \quad n = 2^k \tag{2}$$

Also,

$$\log n = k \log 2$$

$$k = \frac{\log n}{\log 2}$$

$$k = \log_2 n = \lg n \tag{3}$$

Using (3), we can rewrite $T(n)$ as:

$$\begin{aligned} T(n) &= n^2 3^n + n^2 3^{\frac{n}{2}} + n^2 3^{\frac{n}{2^2}} + ... + n^2 3^{\frac{n}{2^{\lg n}}} \\ &= n^2 3^n + n^2 3^{\frac{n}{2}} + n^2 3^{\frac{n}{2^2}} + ... + n^2 3^{\frac{n}{n^{\log_2 2}}} \\ &= n^2 3^n + n^2 3^{\frac{n}{2}} + n^2 3^{\frac{n}{4}} + ... + n^2 3^{\frac{n}{n}} \\ &= n^2 (3^n + 3^{\frac{n}{2}} + 3^{\frac{n}{4}} + ... + 3) \end{aligned}$$

Now we can compute the asymptotic lower bound (Ω) for $T(n)$, as follows:

$$\begin{aligned} T(n) &\geq n^2 3^n \\ \therefore\ T(n) &\in \Omega(n^2 3^n) \end{aligned}$$

Using (2), we can substitute n by 2^k in the exponents of the series in (1) to get,

$$T(n) = n^2 (3^{\frac{2^k}{2^k}} + 3^{\frac{2^k}{2^{k-1}}} + ... + 3^{\frac{2^k}{2^2}} + 3^{\frac{2^k}{2^1}} + 3^{\frac{2^k}{2^0}})$$

$$\implies T(n) = n^2 (3 + 3^2 + ... + 3^{2^{k-2}} + 3^{2^{k-1}} + 3^{2^k}) \tag{4}$$

We can prove that the series $(3 + 3^2 + ... + 3^{2^k-2} + 3^{2^k-1} + 3^{2^k}) < 3^{(2^k+1)}$ as follows:

i. We begin with the following inequality which has to be proved:

$$\begin{aligned} & 3 + 3^2 + 3^4 + ... + 3^{2^{k-1}} + 3^{2^k} < 3^{(2^k+1)} \\ \implies\ & 3 + 3^2 + 3^4 + ... + 3^{2^{k-1}} + 3^{2^k} < 3 * 3^{(2^k)} \end{aligned}$$

ii. Using mathematical induction, we start with the base case, i.e. $k = 0$:

$$\begin{aligned} 3^{2^0} &< 3 * 3^{2^0} \\ 3 &< 3 * 3 \\ 3 &< 9 \end{aligned}$$

iii. As, the inequality holds for $k = 0$, we repeat the process for $k = 1$:

$$\begin{aligned} 3^{2^0} + 3^{2^1} &< 3 * 3^{2^1} \\ 3 + 3^2 &< 3 * 3^2 \\ 12 &< 27 \end{aligned}$$

iv. We see that, the inequality holds for $k = 1$ as well, by induction hypothesis, assume that the inequality holds for $k = l$, l being any arbitrary positive integer, we get the following inequality:

$$3 + 3^2 + 3^4 + ... + 3^{2^{l-1}} + 3^{2^l} < 3 * 3^{(2^l)}$$

v. Now we prove that the inequality holds for $k = l + 1$, as follows:

$$\begin{aligned}(3 + 3^2 + ... + 3^{2^l}) + 3^{2^{l+1}} &< (3 * 3^{2^l}) + 3^{2^{l+1}} \\ &< 3 * 3^{2^{l+1}}(\frac{1}{3^{2^l}} + \frac{1}{3})\end{aligned}$$

We can substitute $(\frac{1}{3^{2^l}} + \frac{1}{3})$ by 1 without violating the inequality, since $(\frac{1}{3^{2^l}} + \frac{1}{3})$ will always be less than 1, given l is positive. This gives us,

$$\begin{aligned}&(3 + 3^2 + ... + 3^{2^l} + 3^{2^{l+1}}) < 3 * 3^{2^{l+1}} * 1 \\ \implies &(3 + 3^2 + ... + 3^{2^l} + 3^{2^{l+1}}) < 3 * 3^{2^{l+1}} \\ &< 3^{2^{l+1}+1}\end{aligned}$$

Hence the inequality is proved. We can multiply, both sides of the general inequality with n^2 to obtain,

$$n^2(3 + 3^2 + 3^4 + ... + 3^{2^{k-1}} + 3^{2^k}) < n^2(3^{(2^k+1)})$$

Using (4), we can replace the LHS of the inequality with $T(n)$,

$$\begin{aligned}T(n) &< n^2(3^{(2^k+1)}) \\ &< 3 * n^2 * 3^{(2^k)}\end{aligned}$$

Using (2), we can replace 2^k by n to get the asymptotic upper bound of $T(n)$,

$$\begin{aligned}T(n) &< 3 * n^2 3^n \\ \therefore\ T(n) &\in O(n^2 3^n)\end{aligned}$$

As, both the lower and upper asymptotic bounds of $T(n)$ are the same, we can write $T(n)$ as:

$$\implies T(n) \in \Theta(n^2 3^n)$$

---

[100] 2. This problem concerns the SUBSET SUM problem.

A. State the SUBSET SUM problem as a language called SS.

B. Describe an efficient encoding scheme for instances of SUBSET SUM. Explain how to encode this instance of SUBSET SUM:

$$\begin{aligned}S &= \{4, 15, 37\} \\ t &= 19.\end{aligned}$$

Give the encoding of the instance as a binary string.

C. Use the dynamic programming paradigm to develop an algorithm SUBSET-SUM-DP to accept SS. Pseudocode is not required.

D. Give the asymptotic worst-case time complexity of SUBSET-SUM-DP.

A. We can define the SUBSET SUM problem as follows:

> SUBSET SUM
> INSTANCE: A set $S = \{s_1, s_2, ..., s_n\}$ of n positive integers and an integer target $t > 0$.
> SOLUTION: A subset $S' \subseteq S$, such that the sum of elements in S' is equal to the target t.

We can state the SUBSET SUM problem as the language:

$$SS = \{e(\langle S, t \rangle) | \text{ there exists a subset } S' \subseteq S \text{ such that } \sum_{s \in S'} s = t\}$$

B. In order to come up with an efficient encoding for instances of SUBSET SUM, we need to encode all the elements of S and the target t in Binary code. We would also need to use Unary code at the beginning of the encoding, to specify the maximum number of bits required to be read, so as to identify t as well as any element in S. This maximum number of bits would depend on the maximum element present in S and the value of t. Let the maximum element in $S = M$, and the binary number representing $M = M_2$. Similarly let the binary number representing $t = t_2$. Now, the number of $1's$ in the Unary code will be equal to $k = |\max\{t_2, M_2\}|$. So, k is the number of bits in the binary encoding of the maximum element $M \in S$, only if $M \geq t$, otherwise this is the number of bits in the binary encoding of t. An encoding for an instance of SUBSET SUM could hence be written as:

$$e(\langle S = \{s_1, s_2, ..., s_n\}, t \rangle) = 111...10\ 01..10\ 00..10\ 10..01\ ...\ 00..01$$

i.e. k 1's followed by a 0 (bits shown in red above), followed by base-2 (binary) code for t (bits shown in green above). This is followed by the base-2 (binary) code for the n elements present in S (shown in blue above). Each binary code will be of length k bits and note that if the total number of bits in the encoding is l, we can easily determine the size n of the set S. This is basically the number of blocks of bits, each of size k that are present after the first $k + 1 + k$ bits ($k + 1$ Unary code for storing k itself and the k-bit long binary code for storing t). So, we can compute the number of elements in S, i.e. n as $n = \frac{l-(2*k+1)}{k}$. Hence, we do not need to keep n as a part of the encoding. We can write the total number of bits (l) required to encode the instance as:

$$\begin{aligned} l &= k + 1 + k + nk \\ &= 2k + 1 + nk \\ &= O(nk) \end{aligned}$$

Here, the space complexity comes out to be $O(nk)$. The following is the encoding for the instance $S = \{4, 15, 37\}$, $t = 19$:

1111110 010011 000010 001111 100101

Here, the maximum element present in S is 37 (which is also greater than $t = 19$) and the binary representation of 37 is 100101, which is 6 bit long, making $k = 6$. Moreover, there are $n + 1 = 3 + 1 = 4$, k-bit binary blocks, with the first block representing t and the each of the subsequent blocks representing an element of S, following the initial $(k + 1)$ bit Unary block.

C. Let S' be the optimal solution for the SUBSET SUM problem, i.e. $S' \subseteq S$, such that $\sum_{s \in S'} s = t$. We can define the value of this optimal solution as $v(n, t)$. Now, using the dynamic programming paradigm, we first need to identify the subproblems of SUBSET SUM, followed by developing a recurrence to get the optimal solution for each subproblem, and then fill a table of optimal values, bottom up using the recurrence. In the end, use backtrace in the table to construct the optimal solution. These four steps for developing the algorithm SUBSET-SUM-DP, are as follows:

(**i**) SUBPROBLEMS: Here, the subproblems are sets of the first i elements in S (i.e. $1 \leq i \leq n$), and j is the target for these sets, such that $j \leq t$. Therefore, a subproblem instance can be written as:

$$\langle S_i = \{s_1, ..., s_i\}, j \rangle$$

And, the value for the optimal solution of a subproblems can be written as, $v[i, j]$.

(**ii**) RECURRENCE: We first look at the two base cases, when $i = 0$, i.e. there are no elements in the set S_i of the subproblem and when $j = 0$, i.e. the target for the subproblem is 0:

$$v[0, j] = 0 \tag{5}$$

$$v[i, 0] = 1 \tag{6}$$

(1), implies that if there are no elements in the set of the subproblem, there can't be any subsets with sum $= j$. (ii), implies that since $\emptyset$ (empty-set) will always be a subset of any S_i, and the sum of elements in $\emptyset = 0$, this will always be $= j = 0$ (as j is the target). The general case, can be written as follows:

$$v[i, j] = \begin{cases} \max \left\{ v[i-1, j - S[i]], v[i-1, j] \right\} & \text{if } S[i] \leq j \\ v[i-1, j] & \text{otherwise} \end{cases}$$

At each step, of the recurrence, we need to take the decision whether to include the i^{th} element or not, given we already have value for the subproblem till the $i - 1^{th}$ element of S. The first case in the general case recurrence, defined for $S[i] \leq j$, implies that given the i^{th} element of S is less than the target j for the subproblem, we have to consider two possibilities for the value of the subproblem:

i. Value of the subproblem upon including $S[i]$ (i.e. $v[i - 1, j - S[i]]$).

ii. Value of the subproblem not including $S[i]$ (i.e. $v[i-1, j]$).

We pick the max of the two values (the maximum of the two values will determine, whether or not to include the i^{th} element of S). The second case in the general case recurrence, applies when the i^{th} element of S cannot be accommodated in the target j, (i.e. $S[i] > j$). In this case the i^{th} element cannot be included in the solution. Hence, the value of our subproblem remains equal to the value for the subproblem till the $i-1^{th}$ element of S (i.e. $v[i-1, j]$).

(**iii**) TABLE: We can now construct the table of optimal values $v[i, j]$, using the defined recurrence, as shown in table 1 below:

$i \downarrow$ / $j \rightarrow$	0	1	2			...			t
0	1	0	0	...	...	...	...	...	0
1	1								
	...								
	...								
	...								
n	1								

Table 1: Table of optimal values for subproblems of SUBSET SUM (each cell of the table, $v[i, j]$ is the optimal value for subproblem with set $S_i = S[1, ..., i]$ and target j).

Here, the rows represent the elements of S, and the columns represent the targets for the subproblems. Using the base cases, the first row would be filled with all 0's except for $v[0, 0]$, which will be set to 1. Also, the rest of the elements in the first column will be set to 1. We can now fill this table from left to right and top to bottom using the general case recurrence defined earlier. All of values in the table would be binary, because we never add or subtract anything from the values set by the base cases which are 0 and 1. If the value at $v[n, t]$ is a 1 (red colored cell), we have an optimal solution for the SUBSET SUM instance for which the table has been made. Consider for example, table 2 below, for the instance of SUBSET SUM given in the question (i.e. $\langle S = \{4, 15, 37\}, t = 19 \rangle$):

$i \downarrow$ / $j \rightarrow$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
3	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1

Table 2: Table for the SUBSET SUM instance $\langle S = \{4, 15, 37\}, t = 19 \rangle$

Here, we see that cell $v[3, 19]$ has the value 1, which implies that there is an optimal solution for this instance. In case there is no solution for the instance the value at $v[3, 19]$ will be 0. So, the SUSBSET-SUM-DP algorithm, can simply

return the value at cell $v[n, t]$ of the table, created for an instance of SUSBET SUM, given as input to the algorithm. We can also find the optimal solution by backtracing.

(**iv**) BACKTRACING: We can look at the value at $v[n, t]$ and bactrace using simple conditions to obtain the optimal solution in linear time. We begin with $S' = \{\}$, now if the value at $v[n, t]$ is 1, we check to see if the element $S[n] \leq t$, if this is true we include the element in the solution (i.e. $S' = S' \cup \{S[n]\}$). Next, we reduce t to account for the n^{th} element, i.e. $t = t - S[n]$ and repeat the check on the $n - 1^{th}$ element. If the element $S[n] > t$, we skip the element (i.e. don't include it in the solution) and repeat the check on the $n - 1^{th}$ element. We keep on repeating this process until we have checked the first element $S[1]$, at this point of time we will have the solution i.e. the set of elements S'. If we apply backtrace on table 2, we get $S' = \{15, 4\}$ (i.e. green colored cells in the table 2).

The pseudocode for SUSBSET-SUM-DP is as follows:

```
SUBSET-SUM-DP(S, t)
 1  let v[0..n, 0..t] be an empty table to be filled.
 2  n = |S|
 3  for i = 0 to n
 4       for j = 0 to t
 5            if (i == 0||j == 0)
 6            // Base cases
 7                 if (i == 0)
 8                      v[i, j] = 0
 9                 if (j == 0)
10                      v[i, j] = 1
11            else
12                 // General case
13                 if (S[i] ≤ j)
14                      if (v[i − 1, j − S[i]] > v[i − 1, j])
15                           v[i, j] = v[i − 1, j − S[i]]
16                      else
17                           v[i, j] = v[i − 1, j]
18                 else
19                      v[i, j] = v[i − 1, j]
20  // Return the value at the bottom-rightmost cell.
21  return v[n, t]
```

SUBSET-SUM-DP, takes as input an instance of SUBSET SUM, constructs the table for the instance, and returns the value at the bottom-rightmost cell of the table. This is equivalent to accepting the language SS, i.e. if there exists a solution for the input instance SUBSET-SUM-DP would return 1 otherwise it will return a 0.

D. The asymptotic worst case time complexity for SUBSET-SUM-DP above, can be computed by looking at the number of times each of its statements would execute. We see

that because of the two **for** loops on lines 3 and 4, the code on lines 5 through 19 will execute $(n+1)*(t+1)$ times. This will result in a worst case time complexity for SUBSET-SUM-DP to be $\Theta(nt)$.

The time complexity depends on the numeric value of t, and therefore SUBSET-SUM-DP may not necessarily run in polynomial time with respect to the input (i.e. n, the number of elements in S).

---

[100] 3. This problem concerns the BALANCED PARTITION problem. In this problem, you have a finite set S of positive integers and K target integers $t_1, t_2, \ldots, t_K$. The solution is a sequence of subsets $S_1, S_2, \ldots, S_K$ of S such that the subsets are pairwise disjoint[1], S is their union[2], and, for $1 \leq i \leq K$, we have

$$\sum_{x \in S_i} x \quad \leq \quad t_i.$$

A. State the BALANCED PARTITION problem as a language BP.

B. Use the NP-completeness proof paradigm to prove that BP is NP-complete.

A. We can define the decision version of balanced-partition problem as follows:

BALANCED PARTITION
INSTANCE: A set $S = \{s_1, s_2, ..., s_n\}$ of n positive integers, and a set $T = \{t_1, t_2, ..., t_K\}$ of target integers, such that $t_i \geq 0$, $\forall\ 1 \leq i \leq K$.
QUESTION: Is there a sequence of subsets $S* = \{S_1, S_2, ..., S_K\}$, such that, $S_i \subseteq S\ \forall\ i,\ 1 \leq i \leq K$, $S_i \cap S_j = \emptyset\ \forall\ i, j,\ 1 \leq i < j \leq K$, $\cup_{i=1}^{K} S_i = S$, and $\sum_{x \in S_i} x \leq t_i$.

A set S is balanced-partitionable into K target integers in a set T, if there exists the sequence of subsets S' (as defined above) for S. We can now define the language corresponding to BALANCED PARTITION as follows:

$$\text{BP} = \{e(\langle S, T \rangle) | S \text{ is balanced-partitionable into the elements of } T\}$$

B. To show that BP is NP-complete, we need to prove:

[1] This means that, for $1 \leq i < j \leq K$, we have $S_i \cap S_j = \emptyset$; note that some S_i may be empty.
[2] That is, $\cup_{i=1}^{K} S_i = S$.

1. BP $\in NP$.

2. BP is $NP - Hard$.

1. BP $\in NP$: A certificate for BALANCED-PARTITION is any sequence of K pairwise disjoint subsets (i.e. S_i's) of S, such that there union is equal to S. We can compute the sum of all elements in each S_i in polynomial time. Next, we can compare each of these K sums to the K integers $\in T$ for equality, in polynomial time. Hence, we can verify BALANCED-PARTITION in polynomial time, and therefore BP $\in NP$.

2. BP is NP-Hard: We can show that SUBSET-SUM $\leq_{\mathrm{p}}$ BALANCED-PARTITION by constructing a reduction function f, which reduces an instance of SUBSET-SUM to an instance of BALANCED-PARTITION in polynomial time.

$$\text{SUBSET-SUM} = \{e(S,t) | S \text{ is a set of positive integers that has a subset } S' \subseteq S, \text{ such that } \textstyle\sum_{s \in S'} s = t\}.$$

f takes $e(S,t)$ (an instance of SUBSET-SUM) and reduces it to $e(\langle S''', T \rangle)$ (an instance of BALANCED-PARTITION). We can construct f as follows:

Let $\langle S, t \rangle$ be an instance of SUBSET-SUM. Let $S''' = S$, $X = \sum_{s \in S} s$ and $T = \{t, X - t\}$ (i.e. $t_1 = t$, $t_2 = X - t$ and $K = 2$). We claim that $e(S''', T)$ is an instance of BALANCED-PARTITION. We prove this claim below.

The sum of all elements in S is X. Since, $\langle S, t \rangle$ is an instance of SUBSET-SUM, there exists a set $S' \subseteq S$, such that the sum of elements in S' is equal to t. Also, there will be another set $S'' = S - S'$, which will contain all the elements in S that are not in S'. Clearly, the sum of elements in S'' will be $X - \sum_{s \in S'} s = X - t$, and the following four properties hold for S' and S'':

i. $S' \subseteq S$, and $S'' \subseteq S$ (both are subsets of S).
ii. $S' \cap S'' = \emptyset$ (both are pairwise disjoint, assuming S is not a multiset)
iii. $S' \cup S'' = S$ (union of the two subsets results in the original set S)
iv. $\sum_{x \in S'} x = t$ and $\sum_{x \in S''} x = X - t$.
(i.e. sum of the elements of the two sets are t and $X - t$ respectively)

In the reduced instance $\langle S''', T \rangle$, as $K = 2$, BALANCED-PARTITION would find two disjoint subsets of S''', i.e. S_1 and S_2, such that the sum of the elements in S_1 and S_2 will be equal to t and $X - t$ respectively. Clearly, $S_1 = S'$ and $S_2 = S''$ because these are the only two subsets of S (and S''', because $S''' = S$) which satisfy all of these properties.

So, given a certificate and instance $\langle S, t \rangle$ for SUBSET-SUM a verifier would return 1, also given a certificate and instance $\langle S''', T \rangle$ a BP verifier would also return 1. Conversely, assume that for the set S in the original SUBSET-SUM instance $\langle S, t \rangle$

there is no subset S' for which $\sum_{x \in S'} x = t$ (consequently, $S'' = S - S'$ also doesn't exist). This would imply that the subsets $S_1 = S'$ and $S_2 = S''$ do not exist for the instance $\langle S''', T \rangle$. So given a certificate and instance $\langle S''', T \rangle$ a BP verifier would return 0 in this case, also given a certificate and instance $\langle S, t \rangle$ a SUBSET-SUM verifier would return 0. Hence, we have reduced an instance of SUBSET-SUM to BALANCED-PARTITION, and since the reduction only involved: 1) Summing up the elements in S to obtain X and 2) Creating the set T consisting of two elements $\{t, X - t\}$, we can be sure that this reduction is a polynomial time reduction. Since, SUBSET-SUM $\in$ NP-Complete and SUBSET-SUM $\leq_{\mathsf{p}}$ BALANCED-PARTITION, we have BALANCED-PARTITION is NP-complete. Hence, the language BP $\in NP - Complete$.